Preprint
UCRL-JC-147847

Terrain Simplification Simplified: A General Framework for View-Dependent Out-of-Core Visualization

P. Lindstrom and V. Pascucci

Shorter version to appear in IEEE Transactions on Visualization and Computer Graphics

U.S. Department of Energy

Lawrence Livermore National Laboratory

May 8, 2002

Terrain Simplification Simplified: A General Framework for View-Dependent Out-of-Core Visualization

Peter Lindstrom and Valerio Pascucci

Abstract—**This paper describes a general framework for out-of-core rendering and management of massive terrain surfaces. The two key components of this framework are: view-dependent refinement of the terrain mesh; and a simple scheme for organizing the terrain data to improve coherence and reduce the number of paging events from external storage to main memory. Similar to several previously proposed methods for view-dependent refinement, we recursively subdivide a triangle mesh defined over regularly gridded data using *longest-edge bisection*. As part of this single, per-frame refinement pass, we perform triangle stripping, view frustum culling, and smooth blending of geometry using geomorphing. Meanwhile, our refinement framework supports a large class of error metrics, is highly competitive in terms of rendering performance, and is surprisingly simple to implement.**

Independent of our refinement algorithm, we also describe several data layout techniques for providing coherent access to the terrain data. By reordering the data in a manner that is more consistent with our recursive access pattern, we show that visualization of gigabyte-size data sets can be realized even on low-end, commodity PCs without the need for complicated and explicit data paging techniques. Rather, by virtue of dramatic improvements in multilevel cache coherence, we rely on the built-in paging mechanisms of the operating system to perform this task. The end result is a straightforward, simple-to-implement, pointerless indexing scheme that dramatically improves the data locality and paging performance over conventional matrix-based layouts.

I. INTRODUCTION

View-dependent refinement and out-of-core data management are two critical components of large-scale, interactive visualization of massive terrain surfaces. In recent years several effective yet quite complicated, often specialized, and many times incompatible methods have been proposed for these two tasks. Whereas large-scale terrain visualization was once synonymous with industrial flight simulation, a plethora of emerging uses, ranging anywhere from military and scientific applications to video games and hobby use, suggest that simple-to-implement yet powerful algorithms for terrain visualization are becoming increasingly valuable. In part to address this problem, we recently proposed a general framework for performing highly interactive view-dependent rendering, as well as a transparent mechanism for improving multilevel cache performance and enabling efficient paging of gigabyte-size data sets [1]. In this paper, we provide an extended overview and in-depth discussion of these algorithms, while taking care to give enough detail to make end-to-end implementations of our algorithms reasonably straightforward.

We will first describe an algorithm for efficient view-dependent refinement. Using the common vertex hierarchy induced by recursive edge/triangle bisection [2–4], we show that it is possible to (1) construct an adaptive mesh from scratch each frame, (2) perform fast, hierarchical view frustum culling, (3) create smooth transitions in the geometry using geomorphing, while (4) simultaneously outputting a single generalized triangle strip for the entire mesh that can be efficiently rendered. Moreover, all of these tasks can be performed without having to maintain any state information, except of course for the output being generated. That is, the traversal can be cast in a purely functional form, which not only makes efficient implementations possible, but is also a feature that meshes well with the out-of-core component of our framework. Because no state information is associated with the mesh vertex data, we can access this data in a read-only fashion. This improves CPU cache performance and also allows the on-disk data to be efficiently *memory mapped* without the need for frequent write-back of dirty pages. As already alluded to, the external memory component of our system is based on associating the on-disk terrain database with a large region of read-only logical address space, which may greatly exceed the amount of physical memory. Under Unix, this can be done using the `mmap` system call, while Windows implementations would use `MapViewOfFile`. Instead of focusing on explicit paging mechanisms, we leave this as an open issue and instead discuss different schemes for rearranging the terrain data so that it can be accessed in a cache coherent manner. We will describe the problem of coherent data layouts in the second part of our paper.

Because our method is stateless, we do not require maintaining dependencies in the vertex hierarchy [2], nor do we make explicit use of frame-to-frame coherence using mechanisms like priority queues [3, 5], active cuts [2, 3, 5, 6], or multi-frame amortized evaluation [6]. We do not mean to imply that such techniques are not useful, however making successful use of these concepts considerably complicates implementations, and we have seen no evidence that our top-down approach cannot perform as well or even better than more complicated previously published methods.

Another feature of our framework is that its individual components are modular—it is, for example, entirely possible to add, remove, or even swap out components such as triangle stripping, culling, geomorphing, data indexing, etc., without having to perform significant code surgery. In addition, adding any one of these components does not change the required on-disk data structures. Rather, the per-vertex terrain data is limited to position (or just elevation), a scalar error term, and a scalar term to encode a bounding sphere. We anticipate that this modularity will aid in quickly implementing the core feature set of our refinement algorithm.

Many of the details of our framework were presented in [1]. We here provide a more thorough exposition, but also significant

The authors are with the Lawrence Livermore National Laboratory, 7000 East Avenue, L-560, Livermore, CA 94551. E-mail: {pl, pascucci}@llnl.gov.

new material. The major contributions of this paper over our previously published research include: (1) An easy-to-integrate technique for position-based geomorphing. The morphs are driven by the screen space projected error for a vertex, which ensures that the terrain geometry is determined entirely by the camera view, and can be varied smoothly with the viewpoint. (2) An extended discussion and derivation of alternative error metrics for use in our framework. (3) Derivations of all index computations needed for hierarchical traversal. (4) A section devoted to a discussion of efficient off-line preparation of the on-disk data. (5) Additional qualitative and quantitative results. We include experimental performance data and animations showing the quality of our geomorphs, and analyze and compare different error metrics. Finally, we have attempted to further clarify the steps in our algorithms to facilitate their implementation and to make the transfer between abstract concepts and actual code as straightforward as possible.

II. PREVIOUS WORK

In this section we discuss related work in large-scale terrain visualization. We will focus particularly on algorithms for view-dependent refinement of terrain, and schemes for out-of-core paging and memory coherent layout of multiresolution data.

A. View-Dependent Refinement

Over the last several decades, there has been extensive work done in the area of terrain visualization and level of detail creation and management. We will here limit our discussion to the more recent work on view-dependent simplification and refinement of terrain surfaces.

Gross et al. [7] were among the first to propose a method for adaptive mesh tessellation at near interactive rates. Their technique is based on a wavelet transform of the gridded data, from which large detail coefficients are chosen for selective refinement. A windowing technique is also described that allows some regions of the mesh to be more refined than others. Lindstrom et al. [2] describe an algorithm for interactive, view-dependent refinement of terrain. They represent the terrain as a mesh with subdivision connectivity that is locally refined using recursive *edge bisection*. The algorithm conceptually works bottom-up, by recursively merging triangles until a screen space error tolerance is exceeded. In actuality, the terrain is partitioned into a quadtree of large rectangular blocks of vertices. Taking advantage of frame-to-frame coherence, the active cut in this quadtree is visited and updated, after which individual vertices within each block are considered for insertion or removal. Due to this blocking of the terrain, special care must be taken to ensure that no cracks form between the blocks. Handling this problem in the context of asynchronous paging of blocks is non-trivial, and enforcing dependencies between vertices can be costly.

Hoppe extended his work on *progressive meshes* to allow view-dependent refinement of arbitrary meshes [6]. This technique was later specialized for terrain rendering [8]. The runtime performance reported by Hoppe places his method among the fastest ones published to date. However, the memory requirements of his method, while lower than in [6], are still considerable. In addition, fully implementing his algorithm is not an easy task.

Using the same space of meshes as in [2], Duchaineau et al. [3] proposed several improvements over Lindstrom et al.'s method in their ROAM algorithm. Instead of organizing the mesh as an acyclic graph of its vertices, they suggest using a binary tree over the set of triangles. Using this data structure, crack prevention is made easier. Another significant contribution is the idea of maintaining two queues for split and merge operations, which allows incremental changes to the mesh to be made in order of importance, while also allowing the refinement to be pre-empted whenever a given time budget is reached. Unfortunately, robustly implementing the dual-queue algorithm, not to mention the many other components of their method, has proven difficult.

Several other algorithms based on edge bisection have since been published, with different strengths and weaknesses in terms of visual accuracy and memory and time complexity [4, 5, 9–11]. These authors recognize the inherent complexity of doing input sensitive bottom-up simplification, and use simple heuristics for output sensitive top-down refinement. Gerstner [11] and Pajarola [4] both discuss how to remove some of the dependencies in the vertex hierarchy by implicitly coding them into the object space errors, but do not extend this concept to view-dependent metrics. Similar to [2], efficient rendering is achieved in [4] by organizing the set of triangles into a single generalized triangle strip that follows the Sierpinski space filling curve. We, too, use a single triangle strip in our refinement algorithm. Röttger [9] presents a memory-efficient solution to terrain rendering, requiring only two bytes of storage per vertex, but his approach relies heavily on a particular view-dependent metric that approximates Euclidean distances with the Manhattan distance. We will revisit some of these methods briefly in the sections below and contrast them with our own method.

B. Geomorphing

The view-dependent level-of-detail algorithms discussed so far have the ability to adapt the terrain mesh at the granularity of individual vertices. Even though this allows fine-scale changes to the mesh to be made from one frame to the next, these changes, if geometrically large enough, can lead to temporal artifacts known as "popping." *Geomorphing*, or just *morphing*, is a common approach to counter such visually disturbing phenomena, by interpolating the geometric transitions between different levels of detail smoothly over time. One downside of morphing is that vertices may have to be introduced earlier than otherwise necessary to allow a continuous transition while still satisfying an error tolerance. However, even without geomorphing the error tolerance is not necessarily set to guarantee subpixel accuracy, but is often specified to be just small enough to eliminate popping. If an error tolerance of several pixels is acceptable, then geomorphing can substantially improve the temporal quality with only a modest computational overhead.

Morphing was first proposed for terrain surfaces by Ferguson et al. [12]. Many view-dependent methods have since incorporated morphing. Cohen-Or et al. [13] proposed using transition zones, based on the distance to each vertex, to blend the geometry of Delaunay-triangulated terrain. Such a distance-based approach was also advocated by Pajarola [4]. Willis and co-

workers described a similar technique that was used in the popular IRIS Performer visual simulation toolkit [14]. Hoppe took a different approach by explicitly animating vertex splits and edge collapses over time. Because of the inherent dependencies between vertices in the hierarchy, his time-based geomorphs imposed somewhat complicated restrictions on when a vertex could be removed. Duchaineau et al. [3] suggested using a similar time-based morphing strategy for their ROAM algorithm.

A slightly simpler and in a sense more disciplined approach than purely time- or distance-based morphing is to make direct use of the given error metric to parameterize the morphs. In this way, the screen space error is used as the parameter that feeds into the interpolation. In the algorithm by Röttger et al. [9], the normalized error term that was used to make refinement decisions was also used as a parameter for morphing the geometry. More recently, Cline and Egbert [15] proposed using a similar approach to morph a quadtree representation of the terrain. For each quadtree patch, they determine a continuous, view-dependent level-of-detail parameter, and use its fractional part to interpolate between the two closest, discrete level-of-detail representations. Our approach to geomorphing is similar in spirit to [9, 15], but we use the actual screen space error as the morph parameter and blend the geometry when this error falls within a user-specified range.

C. Out-of-Core Paging and Data Layout

External memory algorithms [16], also known as out-of-core algorithms, address issues related to the hierarchical nature of the memory structure of modern computers (fast cache, main memory, hard disk, etc.). Managing and making the best use of the memory structure is important when dealing with large data structures that do not fit in the main memory of a single computer. New algorithmic techniques and analysis tools have been developed to address this problem, e.g. for geometric algorithms [17–19] and scientific visualization [20, 21].

In most terrain visualization systems [2–4, 8, 22–26] the external memory component is essential for handling real terrain and GIS databases. Hoppe [8] addresses the problem of constructing a progressive mesh of a large terrain using a bottom-up scheme, by decomposing the terrain into square tiles that are merged after independent decimation, and which are then further simplified. Döllner et al. [27] address the issue of external memory handling of large textures for terrain visualization. Reddy et al. [24] implemented a custom VRML browser specialized for terrain visualization, where efficiency is gained by combined use of multiresolution tiling, data caching, and predictive pre-fetching. The out-of-core component of the large-scale terrain system presented by Pajarola [4] is based on a decomposition of the domain into square tiles, which are stored in a database that supports fast 2D range queries.

Whereas the prevailing strategy for terrain paging has been to split the terrain up into large rectangular tiles of varying resolution that are paged in on demand, and to optimize the size of these tiles and the I/O path from disk to memory, our approach is instead to optimize the data layout to improve the memory coherency—both in-core and out-of-core—for a given access pattern. This approach is in a sense orthogonal to the manner in which the data is paged in. For simplicity, we leave it to the operating system to perform this task.

For accessing large data sets, data layouts based on space filling curves [28] are often used to guarantee good geometric locality [29–31]. To this end, the most popular curve is the Hilbert curve [32], which guarantees the best geometric locality properties [33]. The pseudo-Hilbert scanning order [34,35] generalizes this scheme to rectilinear grids that have a different number of samples along each coordinate axis.

Recently Lawder [36] explored the use of different kinds of space filling curves to develop indexing schemes for data storage layout and fast retrieval in multi-dimensional databases. Balmelli [37] uses the Z-order space filling curve to efficiently navigate a quadtree data structure without using pointers. He uses simple expressions for computing neighbor relations and nearest common ancestors between nodes, allowing fast generation of adaptive edge bisection triangulations. The use of the Z-order space filling curve for traversal of quadtrees [38] (also called Morton-order) has also proven useful in the speedup of matrix operations, allowing better use of the memory cache hierarchies [39–41].

Recently Pascucci [42] introduced a simple address transformation that turns a single-resolution indexing scheme into a multiresolution version, which is optimized for coarse-to-fine breadth-first traversal. This technique has been proven effective for visualizing very large 3D rectilinear grids [43]. The downside of this scheme is the need to apply the address transformation for each data access. The data layout schemes developed in this paper are inspired by this technique, but have more stringent performance requirements. In particular, to achieve high performance, we cannot afford to perform the full address transformation for each data access, and show how to speed the address computation up based on context. Another hierarchical address computation for gridded data was introduced by Gerstner [11]. In this work the bintree hierarchy of triangles induced by the Sierpinski space filling curve is used to fairly efficiently compute the vertex indices during run-time traversal of the data. The locality of the index is inherited from the Sierpinski curve. The application of this address to our case does not seem appropriate because of the scattered set of unused addresses that results from duplicating vertex addresses.

In this paper, we present three different pointerless hierarchical data layouts that have shown to improve the cache and paging efficiency by orders of magnitude over more naive layouts. We draw upon previous work on quadtree and space filling curve layouts, but leverage the fact that the data access pattern is given by a top-down recursive traversal of the height field vertices. We will begin by explaining how this recursive traversal is used for constructing adaptive meshes at run-time.

III. VIEW-DEPENDENT REFINEMENT

The goal of view-dependent level-of-detail algorithms is to construct a mesh with a small number of triangles that for a given view is a good approximation of the original, full-detail mesh. This construction is done continuously at run-time, and whenever the viewpoint changes the mesh is updated to reflect the change. To measure how well the coarse mesh approximates the original, it is common to measure the *object-space* error ϵ between the original mesh and its approximation, e.g. as the

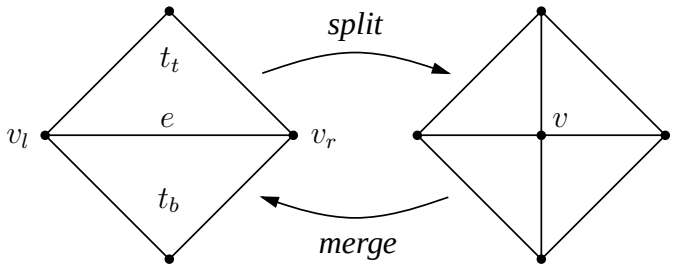


Fig. 1. The split edge $e = \{v_l, v_r\}$ and diamond $T = \{t_b, t_t\}$ of a vertex v.

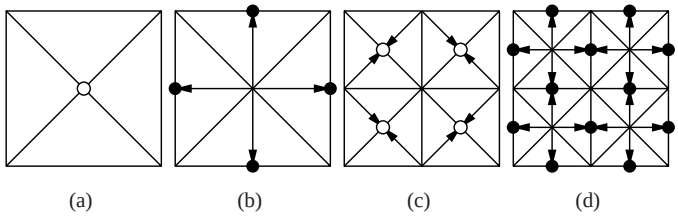


Fig. 2. Edge bisection hierarchy. The arrows correspond to parent-child relationships in the directed acyclic graph of mesh vertices.

vertical deviation between corresponding points, and to project this error onto the screen, e.g. using perspective projection, to obtain a view-dependent measure of error $\rho(\epsilon)$. Depending on whether the mesh is *simplified* bottom-up (fine-to-coarse) or *refined* top-down (coarse-to-fine), triangles are merged or split to ensure that the projected errors meet some tolerance or the mesh meets a given triangle budget.

As is common in terrain visualization, we assume that the input to our refinement algorithm is a terrain surface represented as a uniformly sampled height field, i.e. a rectangular grid of elevations. Formally the height field can be represented as a function $z(x, y)$ over the 2D domain $(x, y) \in \mathbb{R}^2$. To form a continuous surface, we use linear interpolation of the height field, which results in a piece-wise linear triangle mesh. By selecting only a subset of the points from the height field, a coarser mesh is obtained. It is this selection in particular that we will be concerned with below.

In this section, we present a framework for performing top-down, view-dependent refinement of the terrain surface. We show how a single procedure can be used to efficiently perform refinement of the connectivity, blend transitions in the geometry using geomorphing, cull the mesh against the view volume, and simultaneously build a single (generalized) triangle strip for the entire mesh. This procedure makes no use of frame-to-frame coherence, but rather builds the mesh from scratch for each individual frame. We first describe our main approach to refinement, and follow with details of how to implement each of its components.

A. Longest Edge Bisection

There are two important classes of meshes used for view-dependent refinement: general, unstructured meshes (sometimes called triangulated irregular networks, or TINs) [8, 13, 44–46], and regular (or semi-regular) meshes with subdivision connectivity [1–5, 9, 11]. Whereas TINs have the potential to represent a surface with fewer triangles for a given error tolerance (see, for example, [5] for a quantitative analysis), the simplicity of regular subdivision hierarchies makes them more appropriate for our purpose.

In our refinement algorithm, we use a particular type of subdivision based on *longest edge bisection* [2, 3, 9, 11]. The meshes produced by this subdivision scheme, also called 4-k meshes [47], right-triangulated irregular networks [5], and restricted quadtree triangulations [4], have the property that they can be refined locally without having to maintain the entire mesh at the same resolution (see Fig. 3, for example). In the edge bisection scheme, an isosceles right triangle is refined by bisecting its hypotenuse, thus creating two smaller right triangles (Fig. 1). For the vertex v inserted in this refinement step, we call the bisected edge the *split edge* e_v of v. The two triangles (or single triangle in the case of split edges on the boundary) that share e_v are called the *diamond* T_v of v [3]. The split edge and diamond are illustrated in Fig. 1. Starting with a coarse base mesh (typically two or four triangles), an adaptive, recursive refinement of the mesh is made (Fig. 2). The refinement criterion, i.e. whether to split an edge by inserting a vertex, is generally based on whether the vertex's diamond approximates the corresponding part of the full-resolution mesh well enough. For *view-dependent* refinement, this criterion also depends on factors such as the position of the viewer relative to the vertex.

As is evident from Fig. 2, the vertices introduced in the subdivision map directly to points on a regular, rectilinear grid. Thus it is natural to use the edge bisection hierarchy as a multiresolution representation for approximating height fields and terrain surfaces. As in other methods based on edge bisection, the dimensions of the underlying grid are constrained to $2^{n/2} + 1$ vertices in each direction, where n is the (even) number of refinement levels.

It is also possible to perform the inverse of refinement—*simplification*—by starting with the highest resolution mesh and recursively merging pairs of triangles that satisfy a simplification criterion. Simplification often results in a larger reduction in mesh complexity than refinement for any given error tolerance. This is because, as a result of processing the mesh from fine to coarse resolution, the decisions as to where and when to simplify it can be made using the most detailed and accurate information available. In contrast, each refinement decision is necessarily based upon a brief summary of a large amount of information, and generally involves conservative error estimates. A significant disadvantage of simplification versus refinement, however, is that its computational complexity depends on the size of the highest resolution mesh, whereas the refinement complexity is linear in the size of the approximating mesh. Therefore, runtime simplification of very large data sets can be prohibitively slow.

The mesh produced by edge bisection can be represented as a *directed acyclic graph* (DAG) [48] of its vertices. A directed edge (i, j) from i to one of its children j in the DAG corresponds to a triangle bisection, in which j is inserted on the hypotenuse and connected to i at the apex of the triangle (Fig. 2). Thus, all non-leaf vertices not on the boundary of the mesh are connected to four children in the DAG and have two parent vertices. Boundary vertices have two children and one parent. For a given

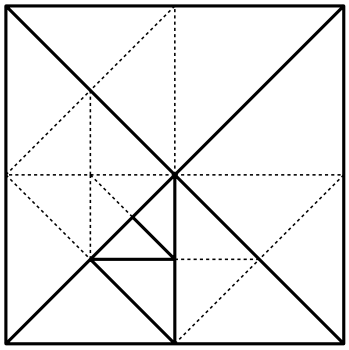


Fig. 3. Example of adaptively refined mesh. In order to avoid cracks in the mesh, the dotted edges must be added.

refinement M of a mesh, we say that a vertex is *active* if it is included in M. Furthermore, M is *valid* if it forms a continuous surface without any T-junctions and cracks. Whether produced by simplification or refinement, for M to be valid it must satisfy the following property:

$$j \in M \Longrightarrow i \in M \qquad j \in C_i \tag{1}$$

where C_i is the set of children of i in the DAG. That is, for a vertex j to be active, its parents (and by induction all of its ancestors) must be active. Fig. 3 illustrates this property, where the dotted edges must be added to form a valid mesh. Even when the DAG traversal is top-down, ensuring this property is not as easy as it may seem, since it is possible to reach j in the DAG without visiting both of its two parents.

One solution to enforcing the validity of the mesh is to maintain explicit dependencies between each child and its parents; whenever a vertex is activated, the chain of dependencies is followed and all ancestor vertices are activated [2]. However, this approach is inefficient both in terms of computation and storage. Our approach, instead, is to satisfy Property 1 by ensuring that the error terms used in the refinement criterion are nested, thereby implicitly forcing all parent vertices to be activated with their descendants.

B. Refinement Criterion

The idea of using nested errors is not new. Pajarola [4] and Gerstner [11] discuss nested object space errors, and refer to the nesting condition as "saturating" the errors. However, neither describe how to guarantee that the errors after projection to screen space remain nested, which as we shall see requires that special care be taken in formulating the error metric. The ROAM algorithm [3] uses nested errors in both object and screen space to order triangles in a priority queue. However, their screen space metric applies only to a restricted class of object space metrics, and assumes that perspective projection is used. In addition, their metric appears considerably more complicated to evaluate than the ones presented in this paper, which in our case is important since we must compute the screen space error for every potentially active vertex in every single frame.

Perhaps the most closely related refinement algorithm to ours is the one proposed by Blow [10]. His method, like ours, is based on a nested sphere hierarchy. Each sphere is centered on the position $\mathbf{p}_i$ of a mesh vertex i, and represents the isocontour of i's projected screen space error $\rho_i = \rho(\epsilon_i, \mathbf{p}_i, \mathbf{e})$, where ϵ_i is an object (or world) space error term for i and $\mathbf{e}$ is the viewpoint.[1] That is, ρ_i is constant for all viewpoints on the sphere's surface. For a fixed screen space error tolerance τ, the isocontour for which $\rho_i = \tau$ divides space into two halves; i is active when the viewpoint is inside the sphere ($\rho_i > \tau$), and inactive for viewpoints outside it ($\rho_i < \tau$). Using these spherical isosurfaces, Blow constructs a forest of nested sphere hierarchies, in which each parent sphere contains its child spheres. The vertices associated with these spheres need not be related in the refinement—as long as the viewpoint is outside a particular sphere, none of the vertices in the sphere's subtree can be active, which allows large groups of vertices to be eliminated quickly.

While theoretically simple, Blow's method has a number of drawbacks. First, to ensure the nesting, τ must be fixed up-front. Second, the method is tied to a particular error metric; a metric based on distance alone. A metric that varies with direction from the vertex to the viewer, such as the one in [2], does not necessarily lead to isosurfaces that have good nesting properties. Third, without maintaining explicit dependencies between vertices, or artificially inflating the spheres wherever necessary, Property 1 will generally not be satisfied, resulting in cracks in the mesh. Finally, every tree in the sphere forest must be visited during refinement. Since this forest can be arbitrarily large, further clustering of the trees may be necessary.

Our approach bears some resemblance to Blow's, but avoids many of these shortcomings. We, too, use a nested DAG of spheres, but for a different purpose, and its structure is given by the relationship between vertices in the refinement. In the discussion below, it is unimportant how the error terms ϵ and ρ are measured—we will discuss possible error metrics later in Section III-C. However, we require that $\rho(\epsilon, \mathbf{p}, \mathbf{e})$ increases monotonically with ϵ when $\mathbf{p}$ and $\mathbf{e}$ are fixed. This is a reasonable requirement; as ϵ increases, we would expect its projection ρ for a given viewpoint to increase as well (or at least remain the same). Using these definitions, a sufficient condition for satisfying Property 1 is

$$\rho(\epsilon_i, \mathbf{p}_i, \mathbf{e}) \geq \rho(\epsilon_j, \mathbf{p}_j, \mathbf{e}) \qquad \forall j \in C_i$$

This is the view-dependent version of the saturation condition mentioned in [49]. To guarantee this property, we could compute an adjusted projected error for i by taking the maximum of ρ_i and ρ_j for all children j. However, we need this relationship to be transitive, meaning that it would have to hold not only for i and its children, but also for all of i's descendants. Visiting every descendant of each active vertex at run-time is clearly impractical for large terrains, since the set of descendants increases exponentially in size. Instead, we compute a conservative bound on ρ_i by making use of our sphere hierarchy.

First observe that ρ_i is made up of two distinct components: an object space error term ϵ_i; and a view-dependent term that relates $\mathbf{p}_i$ and $\mathbf{e}$. Our approach is to separate the two and guar-

[1] In the remainder of this paper, we assume that the generic screen space error ρ_i is a function of the position of i and the viewpoint. Some error metrics may measure error at points other than the vertex positions (e.g. over entire triangles [3, 8]), and may depend on additional view information (e.g. gaze direction [3]). It should be straightforward to generalize our definitions to such error metrics.

antee a nesting for each term. Let

$$\epsilon_i = \begin{cases} \hat{\epsilon}_i & \text{if } i \text{ is a leaf node} \\ \max\{\hat{\epsilon}_i, \max_{j \in C_i}\{\epsilon_j\}\} & \text{otherwise} \end{cases} \quad (2)$$

where $\hat{\epsilon}_i$ is the actual (not necessarily nested) geometric error measured by the object space metric. Then clearly $\epsilon_i \geq \epsilon_j$ for $j \in C_i$. Due to the monotonic relationship between ρ_i and ϵ_i, we must have $\rho(\epsilon_i, \mathbf{p}_i, \mathbf{e}) \geq \rho(\hat{\epsilon}_i, \mathbf{p}_i, \mathbf{e})$, which ensures that there is no loss in visual accuracy. We don't necessarily have $\rho(\epsilon_i, \mathbf{p}_i, \mathbf{e}) \geq \rho(\epsilon_j, \mathbf{p}_j, \mathbf{e})$ for $j \in C_i$, however, since an error projected from $\mathbf{p}_j$ may be arbitrarily larger than an error projected from $\mathbf{p}_i$ (e.g. the viewpoint may be close to $\mathbf{p}_j$ but far from $\mathbf{p}_i$). Therefore, it is not sufficient to nest the object space errors alone, but we must also account for this spatial relationship between parent and child vertices. A naive approach would be to compute the projection of ϵ_i not only from $\mathbf{p}_i$ but from the position of each of its descendants, and then let ρ_i be the largest projection. That is, we would compute the projection from a set of points P_i, where

$$P_i = \{\mathbf{p}_i\} \cup \bigcup_{j \in C_i} P_j$$

For the same reason as above, this is impractical because we would have to visit all descendants of i. Instead, we resort to a more easily expressed superset of points to project from, defined by a ball $B_i \supseteq P_i$ of radius r_i centered on $\mathbf{p}_i$:

$$B_i = \{\mathbf{x} : \|\mathbf{x} - \mathbf{p}_i\| \leq r_i\}$$

The radius r_i of B_i is then

$$r_i = \begin{cases} 0 & \text{if } i \text{ is a leaf node} \\ \max_{j \in C_i}\{\|\mathbf{p}_i - \mathbf{p}_j\| + r_j\} & \text{otherwise} \end{cases} \quad (3)$$

Then $B_i \supseteq B_j$ for $j \in C_i$, i.e. the ball hierarchy is nested. A 2D example of this nesting is shown in Fig. 4. Finally, we define the maximum projected error as

$$\rho_i = \rho(\epsilon_i, B_i, \mathbf{e}) = \max_{\mathbf{x} \in B_i} \rho(\epsilon_i, \mathbf{x}, \mathbf{e})$$

Because $\epsilon_i \geq \epsilon_j$, $B_i \supseteq B_j$, and ρ is monotonic, we must have $\rho_i \geq \rho_j$ for $j \in C_i$. Consequently, if j is active, then so is its parent i, which is what we set out to show.

To compute ρ_i at run-time, we need to perform a constrained optimization over the ball B_i. Because most projection operators ρ are such that they do not have isolated maxima in $\mathbb{R}^3$, except possibly at the viewpoint, $\nabla\rho$ is generally non-zero everywhere, and the maximum of ρ occurs on the boundary of B_i. Nevertheless, finding this maximum may seem like an expensive process. However, it is generally easy to find a simple, closed form expression for the maximum, and we will see in Section III-C how two different metrics can be expressed very concisely. It is interesting to note that this approach to computing conservative error bounds is similar to the strategy used by Lindstrom et al. [2], in which an optimization over nested bounding boxes is done for coarse-grained simplification and refinement of large blocks of vertices.

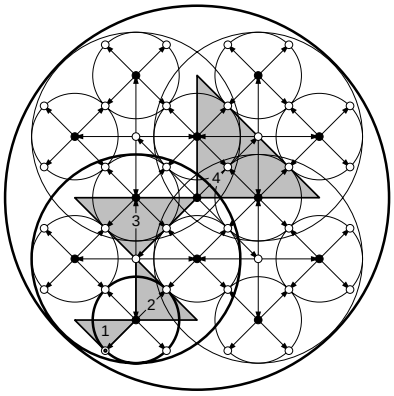


Fig. 4. 2D analogue of the nested sphere hierarchy used for refinement and view culling. The four triangles are associated with the vertices at their right-angle corners. Notice that the bounding spheres do not completely contain their corresponding triangles on the bottom two levels in the DAG, but do contain them on level 3 and above.

During pre-processing of the data set, we compute ϵ and r for each vertex, as described in further detail in Section V. In addition to the vertex's elevation z (and (x, y) coordinates in the domain, if so desired), these are the only parameters needed in our top-down refinement algorithm. We again point out that we have so far left the choice of object space and screen space error metric entirely open. Given this general framework for refinement, we will now briefly discuss how to compute actual screen space errors for different error metrics.

C. Error Metrics

In this section, we consider possible object space (ϵ) and screen space (ρ) error metrics. Typically, the screen space metric is defined in terms of a projection operator, i.e. the screen space error equals the projection of the object space error, and it is often useful to treat the two metrics independently. Our framework is general enough to accommodate virtually any combination of error metrics, which will be illustrated in the following sections by a small set of examples.

C.1 Object Space Error Metrics

Perhaps the most common object space error measure for height fields is the vertical distance between corresponding points in the original and the approximating mesh. For simplicity, these errors are often computed at the height field vertices only [2, 9], but may be computed over triangles or even larger regions of influence associated with a vertex [3, 8]. Our framework accommodates both of these approaches, since the position or region over which the object space error is measured can always be included in a vertex's bounding sphere by inflating it wherever necessary.

Object space errors can also be measured incrementally, between two consecutive levels of refinement [2], or as the maximum error with respect to the highest resolution mesh [3, 8]. The incremental error for a vertex is a good indicator of how much the mesh would *change* by removing the vertex, which may be a useful measure for estimating temporal artifacts due to "popping" (see Section III-F). The maximum error, on the other hand, is a bound on how far the mesh would deviate from the

highest resolution surface if the vertex were removed.

Formally, we write the incremental and maximum vertical errors for a vertex i in terms of the set of triangles T_i in the diamond of i (Fig. 1), i.e. the triangles that share i's split edge. Let $z_t(x_i, y_i)$ be the elevation of triangle t at the point (x_i, y_i) in the domain where vertex i lies. Define the vertical error between i and t as

$$\hat{\delta}_{i,t} = |z_i - z_t(x_i, y_i)|$$

The incremental error can then be written as

$$\hat{\epsilon}_i^{inc} = \max_{t \in T_i} \{\hat{\delta}_{i,t}\} = \left| z_i - \frac{z_l + z_r}{2} \right| \tag{4}$$

That is, the incremental error is the vertical displacement from i to the midpoint of its split edge $\{v_l, v_r\}$. The maximum error can similarly be written by considering i and all of its descendants:

$$\hat{\epsilon}_i^{max} = \max \left\{ \hat{\epsilon}_i^{inc}, \max_{t \in T_i} \max_{j \in D_{i,t}} \{\hat{\delta}_{j,t}\} \right\} \tag{5}$$

where $D_{i,t}$ is the set of all descendants of i reached via recursive bisection of triangle t. Thus, the maximum error is the largest vertical distance between i and its descendants to the two triangles in i's diamond. Note that the measured errors $\hat{\epsilon}_i^{inc}$ and $\hat{\epsilon}_i^{max}$ are not necessarily nested,[2] although $\hat{\epsilon}_i^{max}$ often is since it accounts for distances to all descendants of i. Finally, because the bounding sphere from Equation 3 already contains $D_{i,t}$, we are ensured that the projection of $\hat{\epsilon}_i^{max}$ is a conservative error bound.

The choice between incremental and maximum errors is orthogonal to our refinement method, but should be made up-front since the errors need to be computed and propagated consistently during pre-processing. We will later present results of using both incremental and maximum errors.

C.2 Isotropic Error Projection

Given an object space measure of error ϵ, a view-dependent algorithm projects ϵ onto the screen, resulting in a screen space error $\rho(\epsilon)$. While perspective projection is most commonly used to render the terrain, it involves problems with singularities and can be somewhat computationally inefficient. Therefore it is common in view-dependent algorithms [2, 3, 8] to substitute the distance along the view direction with the Euclidean distance

$$d = \|\mathbf{e} - \mathbf{p}\|$$

between the viewpoint $\mathbf{e}$ and the vertex position $\mathbf{p}$. The most simple metric of this form can be written as

$$\rho(\epsilon, \mathbf{p}, \mathbf{e}) = \lambda \frac{\epsilon}{\|\mathbf{e} - \mathbf{p}\|} = \lambda \frac{\epsilon}{d} \tag{6}$$

i.e. the projected error decreases with distance from the viewpoint. This is an *isotropic* error measure, in the sense that the projected error is the same in every direction a fixed distance d from the vertex. For the usual perspective projection onto a plane, $\lambda = \frac{w}{2 \tan \varphi/2}$, where w is the number of pixels along the field of view φ. Equation 6 is in actuality a projection onto a sphere and not a plane, so a more appropriate choice is $\lambda = \frac{w}{\varphi}$. We then compare ρ against a user-specified screen space error tolerance τ.

In our refinement procedure, we need to find the maximum projection $\rho(\epsilon, B, \mathbf{e})$ over a set of points B (Section III-B). For Equation 6 the maximum projection occurs where $d = \|\mathbf{x} - \mathbf{e}\|$ is minimized. For viewpoints inside B, this term is zero, and we activate the vertex. If $\mathbf{e} \notin B$, then the minimum is $d - r$, and our maximum screen space error becomes

$$\rho(\epsilon, B, \mathbf{e}) = \max_{\mathbf{x} \in B} \rho(\epsilon, \mathbf{x}, \mathbf{e}) = \lambda \frac{\epsilon}{d - r} \tag{7}$$

Comparing ρ against τ and rearranging and squaring some terms (to avoid costly square roots), we obtain

$$\begin{aligned} active(i) &\iff \rho(\epsilon_i, B_i, \mathbf{e}) > \tau \\ &\iff \lambda \frac{\epsilon_i}{d_i - r_i} > \tau \\ &\iff \frac{\lambda}{\tau} \epsilon_i > d_i - r_i \\ &\iff (\nu \epsilon_i + r_i)^2 > d_i^2 \end{aligned} \tag{8}$$

where $\nu = \frac{\lambda}{\tau}$ is constant during each refinement. For spherical projection, $\kappa = \frac{1}{\nu} = \frac{\tau}{\lambda}$ is the angular error threshold in radians. The above expression involves only six additions and five multiplications, and is therefore very efficient to evaluate.

In a strict mathematical sense, the derivation of Equation 8 is valid only if our assumption $\mathbf{e} \notin B_i$ holds. However, if we use the convention that $\mathbf{e} \in B_i \implies active(i)$, then Equation 8 can correctly be used for all viewpoints, with one caveat: If $\epsilon_i = 0$, then its projection ought also be zero, regardless of where the viewpoint is, and the vertex arguably should be deactivated. Of course, we could explicitly test for the special case $\epsilon_i = 0, \mathbf{e} \in B_i$ if it is considered important.

C.3 Anisotropic Error Projection

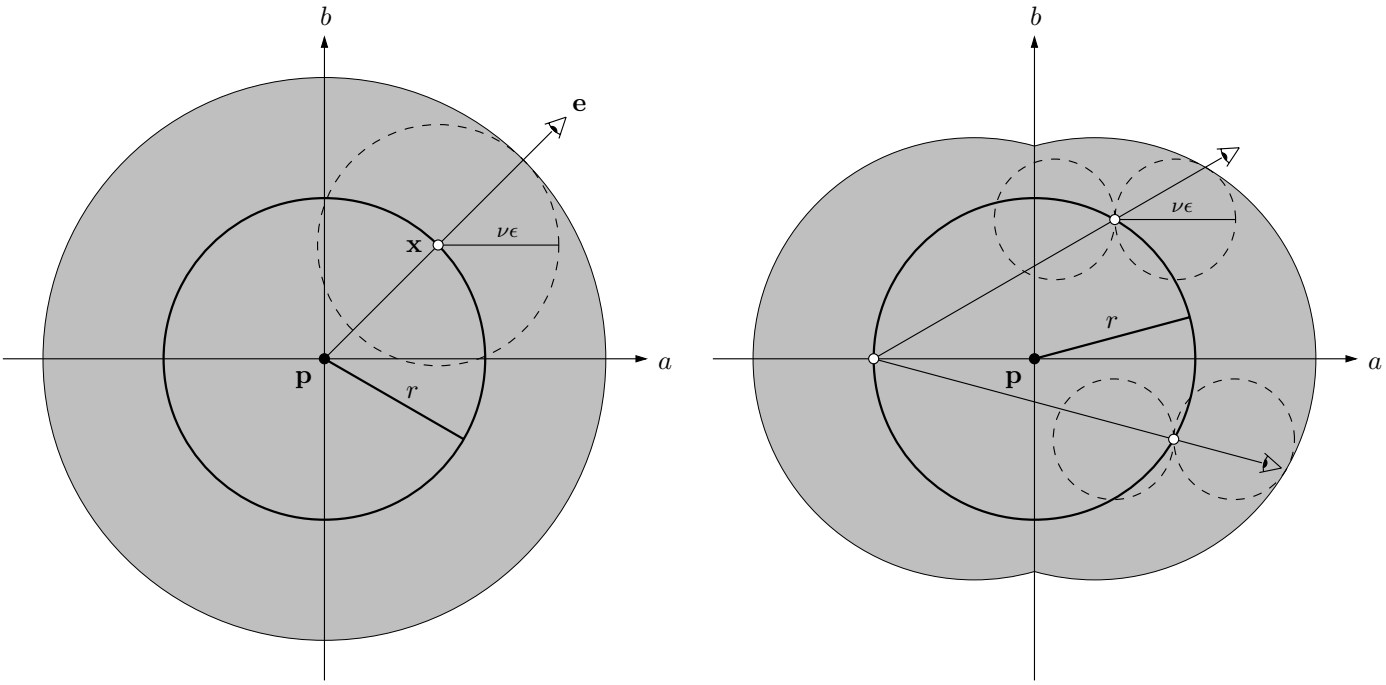
If object space errors are measured vertically, then errors viewed from above appear relatively smaller than errors viewed from the side. As a consequence, vertices directly below the viewer can often be eliminated. Lindstrom et al. [2] describe an *anisotropic* metric $\vec{\rho}$ that exploits this fact. While this metric leads only to marginally fewer triangles, we will here describe how to incorporate it into our framework for illustrative purposes. This metric fundamentally depends on the horizontal and vertical components a and b, respectively, of d:

$$\begin{aligned} a^2 &= (e_x - p_x)^2 + (e_y - p_y)^2 \\ b^2 &= (e_z - p_z)^2 \end{aligned}$$

and we can work with $\vec{\rho}$ in two dimensions to simplify matters. Using these conventions, the anisotropic screen space metric can be written as:

$$\begin{aligned} \vec{\rho}(\epsilon, \mathbf{p}, \mathbf{e}) &= \lambda \frac{\epsilon \sqrt{(e_x - p_x)^2 + (e_y - p_y)^2}}{\|\mathbf{e} - \mathbf{p}\|^2} \\ &= \left(\lambda \frac{\epsilon}{d}\right) \left(\frac{a}{d}\right) \\ &= \rho(\epsilon, \mathbf{p}, \mathbf{e}) \cos\theta \end{aligned} \tag{9}$$

[2] Because refinement generally changes the mesh geometry, it is possible for ϵ^{max} to *increase* from one level to the next as a result of inserting a vertex.



(a) Isotropic projection. (b) Anisotropic projection.

Fig. 5. 2D geometric illustration of isotropic and anisotropic error projection. (a) The point $\mathbf{x}$ in the ball $B = (\mathbf{p}, r)$ where the projection of the object space error ϵ is maximized is the point closest to the viewpoint. Equivalently, $\mathbf{x}$ is the intersection between the ball boundary and a line segment from the viewpoint $\mathbf{e}$ to the ball center $\mathbf{p}$. The dashed circle is an isocontour $\rho(\epsilon, \mathbf{x}, \mathbf{e}) = \tau$ of the screen space error for fixed ϵ and $\mathbf{x}$. That is, the error ϵ projected from $\mathbf{x}$ is constant for all viewpoints on the dashed circle. The shaded region indicates the set of viewpoints for which the vertex at $\mathbf{p}$ is active. This set equals the Minkowski sum of the ball B and the interior of the isocontour. (b) The maximum projection is found as the intersection between the ball boundary and a line segment from the viewpoint to the point opposite the b-axis where the ball meets the a-axis. The isocontours, indicated by dashed lines, for two error maxima are shown. As in (a), the shaded activation region is expressed as the Minkowski sum of the ball and the isocontour.

where θ is the angle of $\mathbf{e}-\mathbf{p}$ above the horizon. As the viewpoint approaches directly above $\mathbf{p}$, θ approaches $\frac{\pi}{2}$ and the projected error vanishes. If on the other hand the viewpoint is at the same elevation as $\mathbf{p}$, then θ is zero and $\vec{\rho}$ equals the isotropic error ρ. Fig. 5(b) shows the isocontours of $\vec{\rho}$ in 2D as being two abutting circles of radius $\frac{1}{2}\nu\epsilon$. The 3D isocontours are tori formed by spinning the circles around their vertical axis of symmetry.

We must now find the maximum $\vec{\rho}$ over all points $\mathbf{x} \in B$. It is relatively easy to show that if $\mathbf{e} \notin B$, then

$$\vec{\rho}(\epsilon, B, \mathbf{e}) = \max_{\mathbf{x} \in B} \vec{\rho}(\epsilon, \mathbf{x}, \mathbf{e}) = \lambda \frac{\epsilon}{d - r} \frac{a + r}{d + r} \qquad (10)$$

The maximum occurs on the boundary of B at a point shown in Fig. 5(b). A simple activation condition associated with $\vec{\rho}_i$ can then be derived as follows:

$$\begin{aligned} active(i) &\iff \vec{\rho}(\epsilon_i, B_i, \mathbf{e}) > \tau \\ &\iff \lambda \frac{\epsilon_i}{d_i - r_i} \frac{a_i + r_i}{d_i + r_i} > \tau \\ &\iff \epsilon_i (a_i + r_i) > \frac{\tau}{\lambda} (d_i^2 - r_i^2) \\ &\iff \epsilon_i a_i > \kappa (d_i^2 - r_i^2) - \epsilon_i r_i \\ &\iff \epsilon_i^2 a_i^2 > \max\{0, \kappa (d_i^2 - r_i^2) - \epsilon_i r_i\}^2 \end{aligned} \qquad (11)$$

assuming $\epsilon_i > 0$ and $\mathbf{e} \notin B_i$. As in the isotropic case, however, this expression can be used unconditionally, and by reusing subexpressions requires at most nine multiplications, seven additions, and one conditional branch.

While the expression for this anisotropic metric is fairly simple, experimental results observed by us and Hoppe [8] indicate that the reduction in mesh complexity over the isotropic metric is only a few percent (sse Section VI). This is mainly because only a small fraction of vertices in a large height field are viewed from above, while the remaining vertices stay active.

D. Run-Time Refinement

Having derived a criterion for selective refinement, we now summarize the algorithm for top-down, recursive refinement and on-the-fly triangle strip construction. Pseudo-code for these steps is listed in Table I. (We will see later how to incorporate view culling and geomorphing into this basic framework.) The refinement procedure builds a generalized triangle strip $V = (v_0, v_1, v_2, \ldots, v_n)$ that is represented as sequence of vertex indices.[3] A vertex v is appended to the strip using the procedure tstrip-append. Line 5 is used to "turn corners" in the triangulation by effectively swapping the two most recent vertices, which results in a degenerate triangle that is discarded by the graphics system [50]. Swapping is done to ensure that the *parity*—whether a vertex is on an even or odd refinement level—is alternating, which is necessary to form a valid triangle mesh. To this end, the two-state variable $parity(V)$ records the parity of the last vertex in V. Fig. 6 illustrates the sequence of triangles traversed during refinement.

[3] An *OpenGL* implementation would make repeated calls to `glVertex` with this sequence of vertices.

```
tstrip-append(V, v, p)
 1 if v ≠ v_{n−1} and v ≠ v_n then
 2     if p ≠ parity(V) then
 3         parity(V) ← p
 4     else
 5         V ← (V, v_{n−1})
 6     V ← (V, v)

submesh-refine(V, i, j, l)
 1 if l > 0 and active(i) then
 2     submesh-refine(V, j, c_l(i, j), l − 1)
 3     tstrip-append(V, i, l mod 2)
 4     submesh-refine(V, j, c_r(i, j), l − 1)

mesh-refine(V, n)
 1 V ← (i_sw, i_sw)
 2 parity(V) ← 0
 3 for each (j, k) ∈ ((i_s, i_se), (i_e, i_ne), (i_n, i_nw), (i_w, i_sw))
 4     submesh-refine(V, i_c, j, n)
 5     tstrip-append(V, k, 1)
```

TABLE I

PSEUDO-CODE FOR RECURSIVE MESH REFINEMENT AND TRIANGLE STRIPPING.

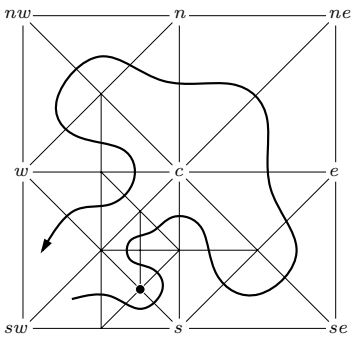


Fig. 6. Traversal of generalized triangle strip. The marked vertex is an example of a situation where swapping is needed in order to pivot around the vertex.

The procedure submesh-refine corresponds to the innermost recursive traversal of the mesh hierarchy, where c_l and c_r are the left and right child vertices of the DAG parent j for the current triangle (Fig. 7). The designations "left" and "right" child do not necessarily correspond to making left and right turns when traversing the DAG. Instead, the sense of left and right alternates between consecutive levels. This is illustrated in Fig. 7, where we have labeled the left and right triangle children for a few levels in the binary triangle tree formed by the edge bisection. This hierarchy extends to DAG vertices by mapping diamonds, i.e. pairs of triangles, to their corresponding vertices (Fig. 1). Notice that if we always follow left branches, we end up in the bottom left corner, whereas following right branches takes us to the bottom right corner. For now, this geometric definition of c_l and c_r is sufficient. We will discuss how to compute these indices numerically from their DAG ancestors i and j in Section IV.

Notice that submesh-refine in Table I is always called recursively with j as the new parent vertex, and the condition

```
submesh-refine(V, i, j, l)
 1 refine ← l > 1 and active(j)
 2 if refine then
 3     submesh-refine(V, j, c_l(i, j), l − 1)
 4 tstrip-append(V, i, l mod 2)
 5 if refine then
 6     submesh-refine(V, j, c_r(i, j), l − 1)
```

TABLE II

EFFICIENT IMPLEMENTATION OF submesh-refine. THE REFINEMENT CONDITION HAS BEEN MOVED UP ONE LEVEL TO AVOID DUPLICATION.

on line 1 is subsequently evaluated twice; once in each subtree. Therefore, most per-vertex work, such as testing for refinement, culling, morphing, etc., is unnecessarily duplicated. Because evaluating the refinement condition constitutes a significant fraction of the overall refinement time, it is more efficient to move it up one level in the recursion, thereby evaluating it only once, and then conditionally making the recursive calls (Table II). For the sake of clarity, however, we will stick to the more concise way (Table I) of writing our recursive functions throughout the remainder of this paper.

Finally, the outermost procedure mesh-refine starts with a base mesh of four triangles (Fig. 2(a)), and calls submesh-refine once for each triangle. Here n is the number of refinement levels, i_c the vertex at the center of the grid, $\{i_{sw}, i_{se}, i_{ne}, i_{nw}\}$ the four grid corners, and $\{i_n, i_e, i_s, i_w\}$ the vertices introduced in the first refinement step (Fig. 6). The triangle strip is initialized with two copies of the same vertex to allow the condition on line 1 in tstrip-append to be evaluated. The first vertex, v_0, is then discarded after the triangle strip has been constructed.

For applications that demand interactive visualization and the highest possible frame rates, it is common to parallelize the otherwise sequential, interleaved tasks of refinement and rendering as two asynchronous processes or threads [14,23]. In this model, the render thread is periodically and asynchronously supplied with a list of geometry to render by the refinement thread. This "display list" is then used, and potentially reused over several frames, until a newly refined mesh is obtained. Our terrain visualization system allows this multi-threaded mode of rendering, in addition to the traditional sequential mode of processing.

E. View Frustum Culling

The rendering performance of our terrain visualization system is substantially improved by culling mesh triangles that fall outside the view volume. Our view culling, which is done as part of the recursive refinement, exploits the hierarchical nature of the subdivision mesh, and culls large chunks of triangles high up in the mesh hierarchy whenever possible. Our approach is based on the culling algorithm outlined in [3], but is somewhat more efficient. In particular, we exploit the nested bounding sphere hierarchy to perform view culling, similar to [6, 51].

Note that the bounding sphere for a vertex i contains the vertices of all descendants of i. Thus, if the bounding sphere is not visible, then neither i nor its descendants will appear on the screen. It is possible, however, for a piece of a triangle t

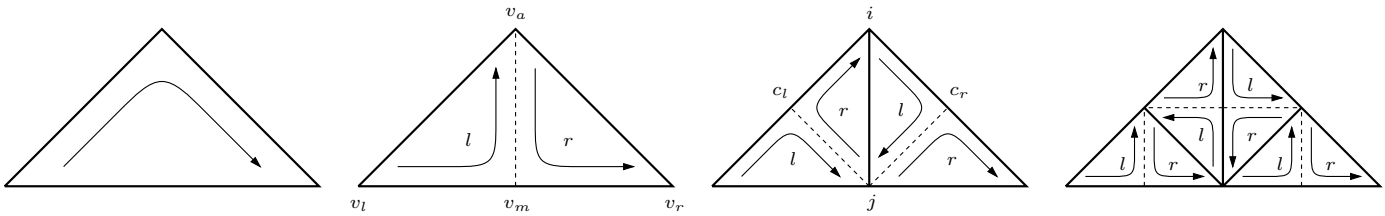


Fig. 7. Binary triangle tree formed by bisection. The arrows indicate the alternating triangle orientation on consecutive levels. The labels l and r correspond to left and right children in the tree, respectively.

visible($i, inside$)
1 **for each** view frustum plane $\langle \hat{\mathbf{n}}_k, d_k \rangle$
2 **if** $\neg inside_k$ **then**
3 $s \leftarrow \hat{\mathbf{n}}_k \cdot \mathbf{p}_i + d_k$
4 **if** $s > r_i$ **then**
5 **return false**
6 **if** $s < -r_i$ **then**
7 $inside_k \leftarrow \mathbf{true}$
8 **return true**

submesh-refine-visible($V, i, j, l, inside$)
1 **if** $inside_k \; \forall k$ **then**
2 submesh-refine(V, i, j, l)
3 **else if** $l > 0$ **and** active(i) **and** visible($i, inside$) **then**
4 submesh-refine-visible($V, j, c_l(i, j), l - 1, inside$)
5 tstrip-append($V, i, l \bmod 2$)
6 submesh-refine-visible($V, j, c_r(i, j), l - 1, inside$)

TABLE III
PSEUDO-CODE FOR VIEW FRUSTUM CULLING.

that has i or one of its descendants as a vertex to be visible, even though none of these vertices are visible. By excluding i, a coarser triangle than t will be rendered. To guarantee that such false positives in the culling test never occur, the bounding sphere for i could be expanded wherever necessary to contain i's incident triangles. Alternatively, the radius of a separate bounding sphere for view culling purposes could be stored with each vertex. In practice, however, the bounding sphere hierarchy is already loose enough that, at least in the domain, the incident triangles are contained above the second finest refinement level (see Fig. 4). Therefore, we have chosen to use the existing hierarchy for view culling, and have seen no visible artifacts of culling the mesh.

The pseudo-code in Table III summarizes our view culling algorithm. The algorithm makes use of the six planes of the view frustum. The parameters $\langle \hat{\mathbf{n}}_k, d_k \rangle$ for each implicit plane equation $s = \hat{\mathbf{n}}_k \cdot \mathbf{x} + d_k = 0$ are computed in object space coordinates and are passed along in the refinement. We ensure that $\hat{\mathbf{n}}$ is a unit length vector so that s is the signed distance to the plane. As in [3], we maintain one flag for each plane, $inside_k$, indicating whether the bounding sphere is completely on the interior side of the plane with respect to the view volume. If this is the case, then all descendants' bounding spheres must also be on the interior side, and no further culling tests against this plane are necessary. If the sphere is on the interior side of all six planes (line 2 of submesh-refine-visible), then we simply transition to our regular refinment procedure without view frustum culling. If on the other hand the sphere is entirely outside any one of the six planes, the vertex and its descendants are culled, and the refinement recursion terminates. Thus, view culling is done only for those spheres that straddle the planes of the view volume.

Fig. 8 illustrates the advantage of performing view culling. From this figure, it is also evident that the mesh resolution drops rather sharply immediately outside the view volume. Still, some features towards the left edge of the mesh in Fig. 8(b) remain, as they are too close to the top plane of the view frustum.

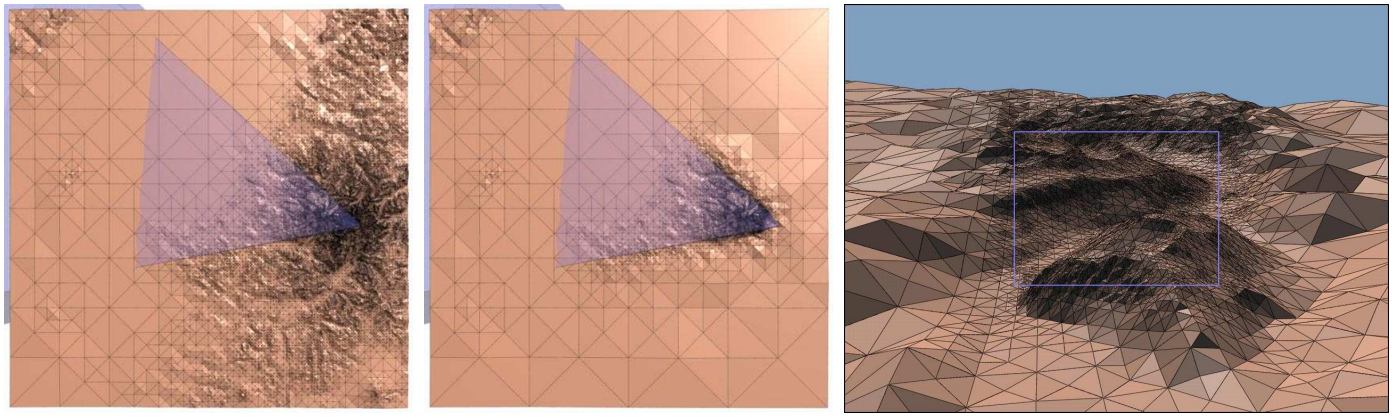
Note that because the bounding spheres are nested, the culling condition is consistent among parents and children, i.e. a child is visible only if its parents are. As a consequence, view culling does not introduce any T-junctions or cracks in the mesh—it always remains a continuous surface everywhere. This is a desirable feature when the refinement and render stages are asynchronous in that, regardless how much the refinement thread falls behind, the render thread always has a continuous mesh to display.

F. Geomorphing

Using our adaptive refinement and view frustum culling algorithms, we can generally maintain high frame rates at no perceptible loss in geometric quality. For example, we typically achieve 60 or more frames per second using a 640×480 window and a two-pixel tolerance. However, for screen resolutions in the mega-pixel range, the tolerance becomes relatively so small that the resulting adaptive meshes can easily exceed 100,000 triangles; too complex even for state-of-the-art graphics hardware to render at these display rates. Increasing the pixel tolerance mitigates this problem, but when τ exceeds a few pixels temporal artifacts become apparent. This is because each vertex insertion results in an instantaneous change in the geometry on the order of τ pixels, and a noticeable and quite disturbing "pop" can be seen near the new vertex. This problem is exacerbated by using flat (per-triangle) shading, in which case the temporal discontinuity in geometry, and hence in surface normals, results in a dramatic change in shading.

A well-known solution to this problem is to use *geomorphing* [12] to smooth out the transitions in geometry over several frames. This is generally done by defining the vertex position as a parametric function $\mathbf{p}(t)$: Whenever a new vertex is inserted, it is initially placed on the current surface ($t = 0$), and is over time slowly moved to its final position ($t = 1$). Conversely, removal of visible vertices is done by reversing the morphing process.

There are two main approaches to geomorphing—time-based



(a) Without culling; 97,435 triangles. (b) With culling; 25,100 triangles. (c) View culling against rectangle.

Fig. 8. Examples of view frustum culling. The mesh is everywhere C^0 continuous, whether culled or not. (a, b) The view frustum is shown in semi-transparent violet, with the viewer looking across the terrain from the right. This view is the same as in Fig. 14. (c) The mesh resolution drops quickly outside the view frustum (shown as a violet rectangle).

and position-based morphing—and they differ mainly in how the parameter t is defined. In time-based morphing [3, 8], the transition occurs over a fixed period of time or number of frames, and vertex positions are typically linearly interpolated over time. The time-based approach can be somewhat difficult to implement, since it requires keeping track of morph start and/or end times for each vertex, and adds additional constraints on vertex dependencies, e.g. one may have to wait for a morph, or even a cascade of morphs, to finish before a vertex can be removed [8].

As its name suggests, position-based morphing [4, 9, 13, 15, 52] uses the position of the viewer instead of time to define the morph parameter t for each vertex. For example, t could be a function of the distance between the viewpoint and the vertex [13]. One advantage of the position-based approach is that both the connectivity and geometry of the adaptive mesh depend only on the position of the viewer, i.e. the mesh always looks the same from any given viewpoint. If in addition $t(\mathbf{e})$ varies smoothly with the viewpoint $\mathbf{e}$, then the mesh geometry is a continuous function of $\mathbf{e}$. Yet another advantage is that, in general, no state information is required to keep track of when the morph was initiated.

Because of its many desirable features, we have chosen to incorporate position-based morphing into our refinement algorithm. We approach this problem by defining a range of pixel thresholds (τ_{min}, τ_{max}), and use morphing whenever the screen space error ρ, which is a function of the viewpoint, falls within this range. We show how this is done for the isotropic (distance-based) error metric discussed in Section III-C.2.

Our goal is to compute the morph parameter t. One possible approach would be to define t as

$$t = \frac{\rho - \tau_{min}}{\tau_{max} - \tau_{min}}$$

Then whenever $t \leq 0$, the vertex is inactive, while $t \geq 1$ implies that the vertex is active and fully morphed. For $0 < t < 1$, we set the elevation z of vertex i to

$$z(t) = tz_i + (1-t)\frac{z_l + z_r}{2} \tag{12}$$

where z_i is the actual, measured elevation of i, and z_l and z_r are the elevations of the endpoints of i's split edge (Fig. 1). Thus, when $t = 0$, i is at the midpoint of the split edge and the geometry is locally no different from when i is absent. Note that z_l and z_r may be the elevations resulting from ongoing morphs for these two vertices, and we may sometimes have a cascading sequence of morphs. Therefore, z_l and z_r must be passed along in the recursion. Fortunately, these vertices have already been visited and their elevations computed higher up in the recursion by the time we reach i.

Using the definition for t above, we have for our isotropic metric

$$t = \frac{\lambda \frac{\epsilon}{d-r} - \tau_{min}}{\tau_{max} - \tau_{min}}$$

This expression involves two divisions and a square root. We can find a simpler (although different) expression by making use of the activation condition in Equation 8. That is, we define a range (d_{min}, d_{max}) for the distance d from the viewpoint to the vertex:

$$d_{min} = \frac{\lambda}{\tau_{max}}\epsilon + r = \nu_{min}\epsilon + r \tag{13}$$

$$d_{max} = \frac{\lambda}{\tau_{min}}\epsilon + r = \nu_{max}\epsilon + r \tag{14}$$

Then, since ρ and d are inversely proportional, $\rho = \tau_{min}$ whenever $d = d_{max}$, and $\rho = \tau_{max}$ whenever $d = d_{min}$. Finally, we define t as

$$t = \frac{d_{max}^2 - d^2}{d_{max}^2 - d_{min}^2} \tag{15}$$

where we have squared the distances to avoid square roots. Thus t, and by extension the vertex position, varies quadratically and smoothly with the distance to the vertex.

Table IV summarizes our geomorphing algorithm for the isotropic metric. Because all distances computed are nonnegative, we can compute their squares directly on lines 1, 2, and 4 in morph. To further improve the performance, we can compute z directly instead of t to avoid any redundant computations when $t = 0$ and $t = 1$. Note that the triangle strip V is no

```
morph(i)
 1 d ← ‖p_i − e‖
 2 d_max ← ν_max ε_i + r_i
 3 if d < d_max then
 4     d_min ← ν_min ε_i + r_i
 5     if d > d_min then
 6         return (d²_max − d²) / (d²_max − d²_min)
 7     else
 8         return 1
 9 else
10     return 0

submesh-morph(V, i, j, l, z_l, z_a, z_r)
 1 if l > 0 and (t ← morph(i)) > 0 then
 2     z ← t z_i + (1 − t) z_a
 3     submesh-morph(V, j, c_l(i, j), l − 1, z_l, (z_l + z_r)/2, z)
 4     tstrip-append-point(V, x_i, y_i, z, l mod 2)
 5     submesh-morph(V, j, c_r(i, j), l − 1, z, (z_l + z_r)/2, z_r)
```

TABLE IV
PSEUDO-CODE FOR GEOMORPHING.

longer a list of vertex indices. Rather, each vertex i in the strip is specified directly by its morphed xyz-coordinates $\langle x_i, y_i, z\rangle$.

Whereas the computations involved in performing geomorphing add to the refinement time, the improved temporal quality of the animation often allows a considerably larger pixel tolerance to be used, which results in far fewer triangles and an overall shorter refinement time. As observed by Hoppe [8] and others, errors as large as several pixels may go unnoticed if geomorphing is used to mask any temporal artifacts. As is evidenced by the accompanying video (see Section VI), a lower threshold τ_{min} as large as six pixels for a 640×480 window can be used.

The temporal quality of the morph generally depends on the length in time over which the morph takes place. If the morph time is too short, then not enough temporal continuity is provided. If on the other hand the morph time is very long, then either τ_{min} must be small, resulting in a high-complexity mesh, or τ_{max} must be large, resulting in a highly inaccurate mesh and a large number of costly morph computations. We experimented with several choices of τ_{min} and τ_{max}, and found that $\tau_{max} = \frac{3}{2}\tau_{min}$ generally provided a good tradeoff between quality and complexity. We used this relationship for the video sequences and results in Section VI. The morph time also depends on the speed at which the viewer is moving. For high flight speeds, the morph time is short. On the other hand, the perceived vertex speed due to viewer motion often outweighs the vertex speed due to morphing, which tends to reduce the effects of short morph times.

IV. DATA LAYOUT AND INDEXING

This section addresses the problem of laying out the terrain data on disk to achieve efficient out-of-core performance. In the spirit of our overall approach to terrain visualization, our goal is to have a very simple mechanism for performing out-of-core paging of the data, while maintaining high performance. In particular, we take advantage of the paging mechanism of the operating system by using the `mmap` system call.[4] `mmap` associates a part of the logical address space of the computer with a specific disk file. Using this mechanism the external memory part of our implementation consists simply of a call to `mmap` to associate the memory address of an array with the terrain information (elevation values, precomputed errors, etc.) stored on disk. After this step the array of terrain vertices is used as if it were allocated in main memory, while the operating system takes care of paging the data from disk as needed.

The main advantage of this approach is its simplicity. Moreover, since the paging mechanism is not specialized for one particular out-of-core algorithm, we can perform a fair comparison among different data layout schemes. In this paper we study the performance potential intrinsic in different data layouts, without adding any specialized I/O layer with pre-fetching mechanisms that might further improve the out-of-core performance of the terrain traversal.

Given the framework described above, the external memory processing problem can be reduced to a data layout problem. That is, we want to find a permutation of the set of mesh vertices such that their layout on disk closely follows the order in which they are typically accessed during refinement. We know the structure of the terrain traversal algorithm, and we have a mechanism that hides the need for data paging from the application. Based on this, we need to determine: (1) a way of storing the raw data that minimizes paging events, and (2) an efficient procedure for computing the index of the data element in the given refinement order, so that no significant added cost is introduced in the refinement process.

The following two subsections describe a data layout scheme that satisfies requirements (1) and (2), and that has a particularly straightforward implementation.

A. Interleaved Quadtrees

On the basis of the edge bisection refinement algorithm, each vertex (apart from the four corners of the grid) can be labeled as white, if introduced at an even level of refinement, or black, if introduced at an odd level. Fig. 2 shows this classification for the first four levels of refinement. The top row of Fig. 9 shows how the sequence of white vertices forms a quadtree—the *white quadtree*, Q_w. Each white node is in fact the center of a square tile in a quadtree decomposition of the rectilinear grid. Interestingly the black vertices can also be considered as part of a *black quadtree*, Q_b. Fig. 9 shows as crossed circles the vertices that need to be added outside the rectilinear grid to form a complete black quadtree. We will refer to these additional vertices as "ghost vertices." The black quadtree is rotated 45 degrees with respect to the white quadtree. Note that Q_b does not start at the root but at the first level of refinement. Adding a virtual root node makes Q_b one level taller than Q_w.

Since the traversal of the DAG (see Section III-A) is performed top-down, starting form the root, good data locality can be achieved by storing the data from coarse to fine levels. Within each level, the data should be stored so as to preserve neighborhood properties to the extent possible; vertices that are geometrically close should be stored close together in memory. For a

[4] The equivalent Windows function is called `MapViewOfFile`.

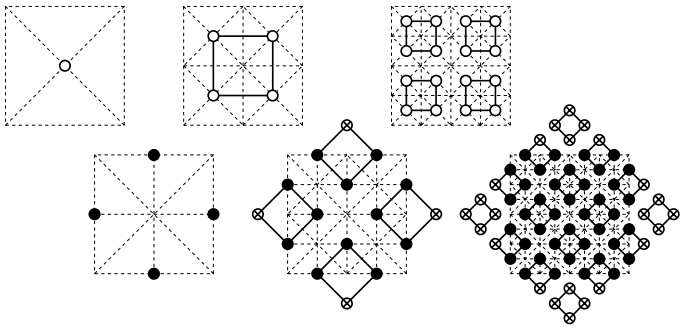


Fig. 9. Top row: First three levels of the white quadtree. Bottom row: A complete black quadtree is obtained by adding the crossed ghost vertices to it.

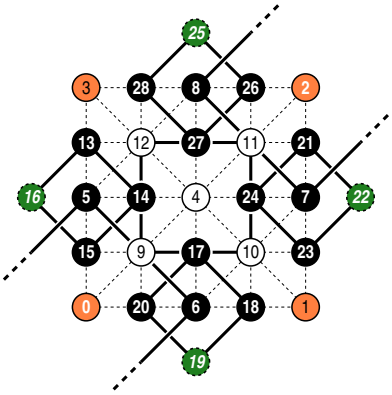


Fig. 10. Indices for the first few levels of the interleaved quadtrees. Ghost vertices are marked in green; the orange corner vertices are not part of either quadtree.

quadtree, this can be achieved by using the order induced by the following formula that computes the index $c(p,k)$ of the k^{th} child of the parent node p:

$$c(p,k) = 4p + k + m \qquad \text{with } k = 0,1,2,3 \qquad (16)$$

where m is a constant dependent on the index of the root and the index distance between consecutive levels of resolution. Using this data layout, all the vertices on the same level of resolution are stored together, starting with the coarsest level. The index distance between two vertices on the same level depends on the distance to their common ancestor in the quadtree, e.g. any four siblings are stored in consecutive positions. For this indexing scheme, we interleave the black and the white quadtree, with roots $r_b = 3$ and $r_w = 4$. Since r_b is not used in practice, we can assign the first four indices (from 0 to 3) to the corners of the grid (Fig. 10). The first child of r_b is stored immediately after r_w, and we have

$$c(r_b, 0) = 4 \cdot 3 + 0 + m = 5$$
$$c(r_w, 0) = 4 \cdot 4 + 0 + m = 9$$

which both imply $m = -7$. Fig. 10 shows the vertex indices for the first few levels of the interleaved quadtrees.

B. Embedded Quadtrees

Notice in Fig. 9 that the ghost vertices in Q_b are not used. Because the data is eventually stored as a single linear array, this

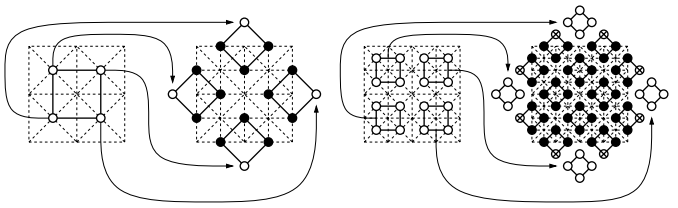


Fig. 11. Illustration of embedding the top two levels of the white quadtree in the unused parts of the black quadtree.

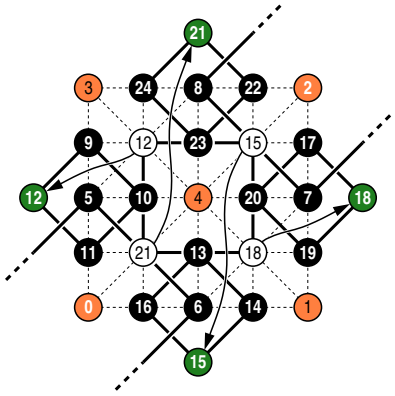


Fig. 12. Indices for the first few levels of the embedded quadtrees.

results in unwanted "holes" in the array. It is however possible to reduce the amount of unused space. First observe that the total number of ghost vertices is roughly twice as large as the number of white vertices. As a consequence, instead of using two interleaved quadtrees, we can use the black quadtree only and store the white nodes in place of (a subset of) the ghost nodes. We divide Q_w into four subtrees, rooted at the children of r_w. Fig. 11 shows the insertion of these subtrees into the unused space of Q_b. The use of a single quadtree also affects the value of the constant m. In this case we have $r_b = 4$ (this value is actually used for the white root) and $c(r_b, 0) = 5$, which implies $m = -11$. In addition, since the white quadtree has been split up into four independent subtrees, these relocated subtrees will not be reached from the white root r_w (node 4 in Fig. 12) using Equation 16. Therefore we cannot begin the recursive refinement with r_w, but must unroll the recursion one level and make eight instead of four calls to submesh-refine from mesh-refine.

C. Efficient Index Computation

To avoid any overhead in the refinement process, we need an efficient method for computing the indices of the vertices visited in our top-down traversal of the terrain. For data stored in linear order (standard row major matrix layout), computing the child indices in the DAG can be made easy by carrying along three indices in the refinement: (v_l, v_a, v_r). These indices make up the current triangle t in the refinement, and their subscripts correspond to the left, apex, and right corner of the triangle (Fig. 7). The two child triangles of t in the recursion can then be written as $t_l = (v_l, v_m, v_a)$ and $t_r = (v_a, v_m, v_r)$. Here v_m corresponds to the vertex at the midpoint of the split edge $\{v_l, v_r\}$, which can be computed simply as the index average $v_m = (v_l + v_r)/2$.

For the indexing scheme based on the interleaved quadtrees,

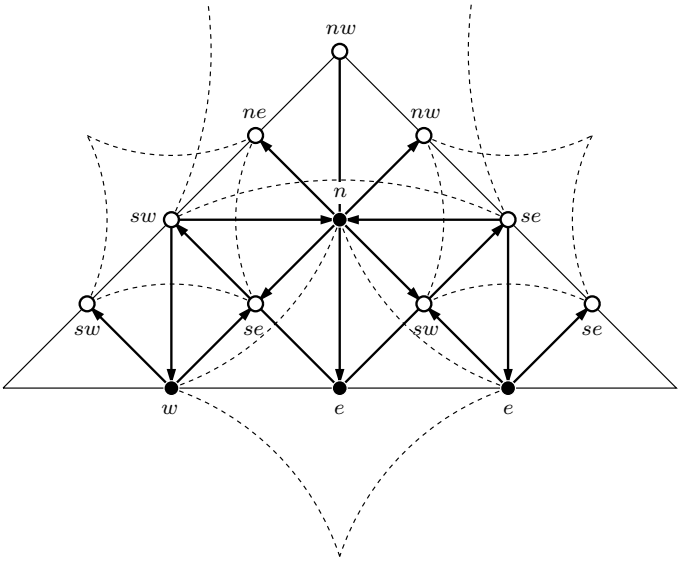


Fig. 13. Quadtree branches taken during DAG traversal. For example, after going from an sw-child p_q in the white quadtree to a w-child p_g in the black quadtree, the two *graph children* of p_g are the sw and se *quadtree children* of p_q. See Table V for all possible branch combinations. The dashed arcs in the figure show quadtree sibling relationships.

we make use of the parent-child relationship between vertices in the quadtrees. Consider one refinement step as shown in Fig. 7. The new white vertices c_l (left child) and c_r (right child) have a common *graph parent* p_g in the refinement DAG (j in the figure). Moreover the graph parent of p_g is also the *quadtree parent* p_q of c_l and c_r (i in the figure). Based on this observation, the indices c_l and c_r can be computed from the index of their quadtree parent p_q using Equation 16. The relative positions of p_q and p_g determine which two branches (the values of the index k) need to be used to reach c_l and c_r from p_q. This relationship is illustrated in Fig. 13 and summarized in Table V. To find what branch k a given child node c corresponds to, we simply solve Equation 16 for k:

$$k = (c - m) \bmod 4 \tag{17}$$

That is, the value of k can be determined from the lowest two bits of the vertex index. We can then use Equation 16 and Table V to compute c_l and c_r. However, there is considerable redundancy in the transition tables, and by carefully numbering the four branches in the two quadtrees it is possible to compute c_l and c_r using simple arithmetic. We show how this is done below.

In order to make the transition tables as simple as possible, we have chosen to number the quadtree branches as follows: $sw = n = 0$, $se = e = 1$, $ne = s = 2$, $nw = w = 3$. That is, the order is counterclockwise in the white quadtree and clockwise in the black quadtree. Because of this choice, Table V can be collapsed to a single table, Table VI, that can be used for both quadtrees. Let us focus on how to encode Table VI using arithmetic. We will use $k(k_q, k_g, b)$ to denote both k_l and k_r, with the convention that b is zero for k_l and one for k_r. First observe that rows 0 and 2 are the same, as are rows 1 and 3, thus

$$k(k_q, k_g, b) = k(k_q \bmod 2, k_g, b)$$

k_l, k_r		k_g			
		n	e	s	w
k_q	sw	se ,ne	ne,nw	nw,sw	sw, se
	se	nw,sw	sw, se	se ,ne	ne,nw
	ne	se ,ne	ne,nw	nw,sw	sw, se
	nw	nw,sw	sw, se	se ,ne	ne,nw

k_l, k_r		k_g			
		sw	se	ne	nw
k_q	n	e , s	s , w	w , n	n , e
	e	w , n	n , e	e , s	s , w
	s	e , s	s , w	w , n	n , e
	w	w , n	n , e	e , s	s , w

TABLE V

TRANSITION TABLES USED TO DETERMINE THE LEFT (k_l) AND RIGHT (k_r) BRANCH TO BE USED IN EQUATION 16 FOR QUADTREE PARENT BRANCH k_q AND GRAPH PARENT BRANCH k_g.

k_l, k_r		k_g			
		0	1	2	3
k_q	0	1, 2	2, 3	3, 0	0, 1
	1	3, 0	0, 1	1, 2	2, 3
	2	1, 2	2, 3	3, 0	0, 1
	3	3, 0	0, 1	1, 2	2, 3

TABLE VI

SINGLE TRANSITION TABLE CORRESPONDING TO $sw = n = 0$, $se = e = 1$, $ne = s = 2$, $nw = w = 3$ (CF. TABLE V).

We can therefore focus only on rows 0 and 1. Now notice that if we shift row 1 two columns to the left, then rows 0 and 1 are the same. That is

$$\begin{aligned} k(k_q, k_g, b) &= k(k_q \bmod 2, k_g, b) \\ &= k\big(0, (2(k_q \bmod 2) + k_g) \bmod 4, b\big) \end{aligned}$$

We are now left with row 0 only, from which we immediately notice that $k_l = (k_q + 1) \bmod 4$ and $k_r = (k_l + 1) \bmod 4$, i.e.

$$\begin{aligned} k(k_q, k_g, b) &= k\big(0, (2(k_q \bmod 2) + k_g) \bmod 4, b\big) \\ &= (2(k_q \bmod 2) + k_g + b + 1) \bmod 4 \\ &= (2k_q + k_g + b + 1) \bmod 4 \end{aligned}$$

Since $k_q = (p_q - m) \bmod 4$ and $k_g = (p_g - m) \bmod 4$, we have

$$\begin{aligned} k_l(p_q, p_g) &= (2(p_q - m) + (p_g - m) + 1) \bmod 4 \\ &= (2p_q + p_g - 3m + 1) \bmod 4 \\ &= (2p_q + p_g + m + 1) \bmod 4 \\ k_r(p_q, p_g) &= (2p_q + p_g + m + 2) \bmod 4 \end{aligned}$$

Finally, we arrive at the following expressions for c_l and c_r:

$$\begin{aligned} c_l(p_q, p_g) &= 4p_q + ((2p_q + p_g + m + 1) \bmod 4) + m \\ c_r(p_q, p_g) &= 4p_q + ((2p_q + p_g + m + 2) \bmod 4) + m \end{aligned}$$

These simple equations are used in the submesh-refine procedure in Section III-D.

D. Memory-Efficient Hierarchical Indexing

One drawback of our quadtree-based indexing schemes is that they use a non-contiguous address space. In the case of interleaved quadtrees, the unused ghost vertices result in a waste in storage resources of roughly 66% of the input data. This overhead is reduced to 33% in the storage layout where the white quadtree is embedded in the black quadtree. This overhead can be completely eliminated by using a data layout based on a hierarchical version of the Π-order space filling curve. Because the implementation of this scheme is not as straightforward as the quadtree-based schemes described above, we have deferred the derivation and discussion of the Π-order layout to Appendix A. In Section VI we include empirical results of the performance achieved both with the quadtree-based schemes discussed here and with the hierarchical Π-order space filling curve.

V. DATA PREPROCESSING

Sections III and IV describe what information needs to be computed for each vertex and how to organize the information. These sections do not, however, provide much detail about the steps involved in preparing the data in this format. We here discuss possible representations of the data and explain a method for preparing the data in an off-line preprocessing phase.

A. Vertex Representation

As discussed in Section III-B, in addition to the elevation z of each vertex, we must store an error term ϵ and a bounding sphere radius r. In order to avoid carrying the xy-coordinates in the domain with us during the recursive refinement, we can optionally store the full position $\mathbf{p} = (x, y, z)$ with each vertex. In our visualization system, we have chosen to store all of the fields as 32-bit floating point numbers. For fixed-length records $\langle \mathbf{p}, \epsilon, r \rangle$, this means 20 bytes of storage per vertex.

B. Bottom-Up Propagation

Given a representation for the vertex records, we now turn our attention to how to compute the individual fields of these records. In particular, we describe how to efficiently compute the error term ϵ (Equation 2) and the bounding sphere radius r (Equation 3). Clearly, because of their recursive definitions, both ϵ and r must be computed and propagated *bottom-up* from DAG children to their parents. To do this, one possible approach would be to traverse the DAG using the recursive refinement procedure from Section III-D, and to propagate the computed values back up the recursion. Unfortunately, using this approach only two children are visited at a time, and the information is propagated to only one parent. (Recall that each vertex has up to four children and two parents in the DAG.) While making repeated recursive traversals would eventually guarantee that all the information is propagated up the DAG, this approach is inefficient. Instead, we describe a method that processes the information level-by-level and visits each vertex only once.

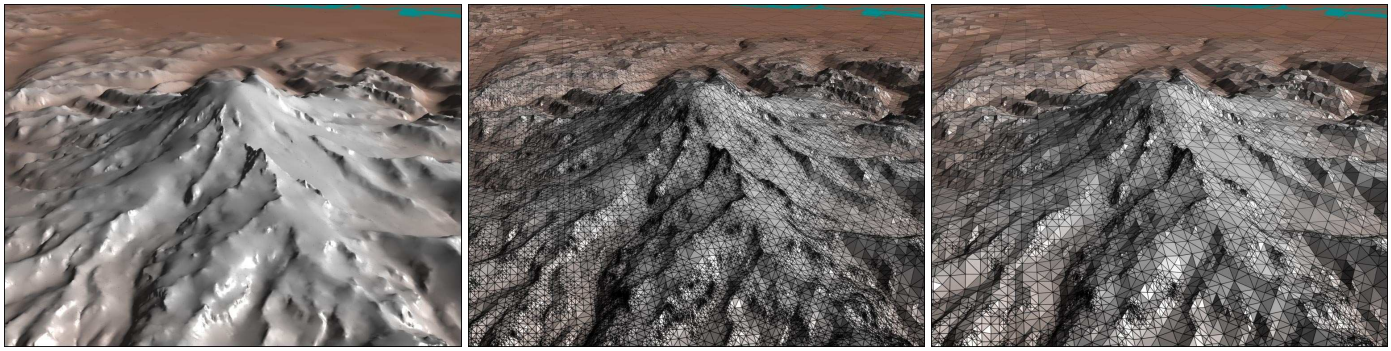
Notice in Fig. 2 that we can relatively easily identify the vertices that fall on any given refinement level. In the white quadtree we always have a single square grid of dimensions $2^n \times 2^n$, whereas in the black quadtree we have two overlapping grids, each with $2^n \times (2^n + 1)$ vertices. For example, in Fig. 2(d) we have a 2×3 grid and a 3×2 grid of black vertices, each being the transpose of the other. If the vertex data is stored in a linear 2D array, then we can traverse the vertices on each level by simply stepping over the appropriate number of vertices in each direction. We can also find simple offset rules to reach the four children from any given vertex, as indicated by the arrows in Fig. 2. Finally, in order to compute the error term ϵ, we may need to identify the split edge associated with each vertex (the remaining two vertex indices in the diamond can be derived similarly, if needed). For the refinement levels containing black vertices, this is easy; the split edges are always oriented in the direction of the grid that has the smaller number of vertices. On white levels, the split edges alternate in orientation from one vertex to the next. Thus, we have a set of simple rules for traversing the vertices on a level and accessing their children, split edges, and diamonds.

In Section IV we described several different data layouts. These layouts do not necessarily lend themselves to the simple level-by-level traversal just described. Rather, we know only how to traverse them recursively. Therefore, we have chosen to perform all preprocessing by first storing the height field in linear, row major order, and then as a final step rearranging the data to fit the given layout. In practice, this ends up being considerably more efficient than performing repeated recursive traversals until the information is fully propagated. This rearrangement of the data is done using two parallel recursive traversals. That is, we simultaneously traverse the linear input array and the reordered output array in refinement order, using the indexing rules from Section IV, and copy corresponding vertices to the output array. We acknowledge that this requires roughly twice as much memory, and assumes that the processing can be done in-core. For extremely large terrains, it may be necessary to memory map the output data, and to work only on small pieces of the input data at a time.

VI. RESULTS

In this section, we present the results of running an implementation of our terrain visualization system on several computer architectures. We used a two-processor 800 MHz Pentium III PC, running Red Hat Linux, with 900 MB of RAM and GeForce2 graphics. To push the out-of-core aspect of our system, we artificially limited the memory configuration of this machine to 64 MB for some of our results. A two-processor 300 MHz R12000 SGI Octane with Solid Impact graphics and 900 MB of RAM was also used to measure memory coherency, while we used a 48-processor 250 MHz R10000 SGI Onyx2 with 15.5 GB of RAM and InfiniteReality2 graphics to avoid being graphics and memory limited, and to allow the raw refinement speed to be measured. For all results, we used a data set over the Puget Sound area in Washington (see Fig. 14), which is made up of $16,385 \times 16,385$ vertices at 10 meter horizontal and 0.1 meter vertical resolution.[5] Using our data structures, this data set occupies roughly 5 GB on disk. The quantitative results presented here were collected during a 2,816-frame fly-over of this data set. The window size was in all cases 640×480 pixels.

[5] This data set can be downloaded from http://www.cc.gatech.edu/projects/large_models/ps.html.



(a) $\tau = 2$ pixels; 79,382 triangles. (b) $\tau = 2$ pixels; 79,382 triangles. (c) $\tau = 4$ pixels; 25,100 triangles.

Fig. 14. View of Mount Rainier, Washington. (b, c) Edge bisection subdivision meshes for two different screen space error thresholds τ.

A. View-Dependent Refinement

We will first discuss the performance of our view-dependent refinement algorithm. We used the distance-based error metric described in Section III-C.2 for the results presented here. To evaluate the efficiency in mesh complexity for a given accuracy, we recorded for the 2,816-frame fly-over the number of rendered triangles obtained using both a bottom-up simplification of the terrain and our conservative top-down scheme. In the bottom-up scheme, the object space errors need not be inflated to guarantee nesting, nor do the projected errors have to be inflated by measuring them over the nested bounding spheres. Instead we used the actual projected error $\rho(\hat{\epsilon}_i, \mathbf{p}_i, \mathbf{e}) \leq \rho(\epsilon_i, B_i, \mathbf{e})$ of the measured error $\hat{\epsilon}_i$ for each vertex, and then performed a separate step to patch cracks by activating the ancestors of all active vertices. Using this bottom-up approach, we obtain for any given metric and tolerance the minimal valid mesh possible,[6] which consequently serves as a good benchmark for evaluating our refinement method. The results presented here are for meshes that have been coarsened using view frustum culling, and differ somewhat from the results presented in [1] where culling was not used.

Fig. 15 shows for both the incremental ($\hat{\epsilon}^{inc}$) and maximum ($\hat{\epsilon}^{max}$) object space metrics the number of triangles produced during the fly-over by bottom-up simplification, as well as the relative percentages of additional triangles produced by our conservative top-down refinement. We used a tolerance $\tau = 1$ pixel and a coarsened version of the Puget Sound data downsampled to 1025×1025 vertices (to make data collection tractable). As can be seen from the shaded regions, the maximum error $\hat{\epsilon}^{max}$ consistently resulted in a small increase in mesh complexity over using $\hat{\epsilon}^{inc}$. This is not surprising since $\hat{\epsilon}^{max} \geq \hat{\epsilon}^{inc}$ (Section III-C.1). However, as can be seen in the graph, the discrepancy is often small, suggesting that $\hat{\epsilon}^{inc}$ can be used to approximate the more compute-intensive $\hat{\epsilon}^{max}$. Because of its simplicity, we chose to use the incremental error for the remaining results in this paper.

The two curves in Fig. 15 show on the rightmost vertical axis

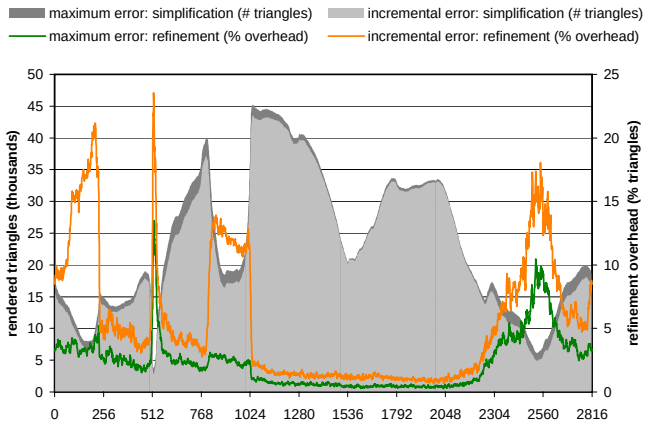
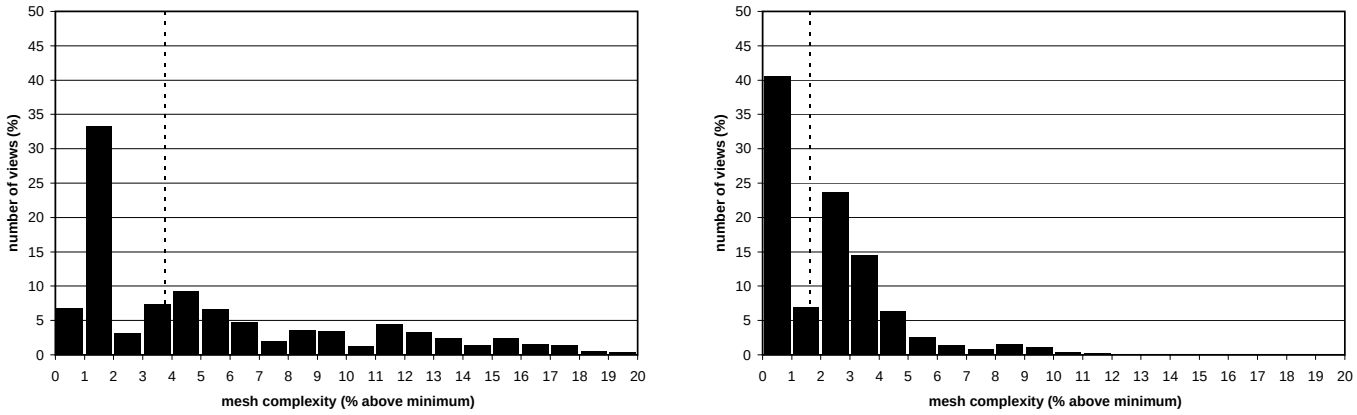


Fig. 15. Mesh complexity comparison between top-down refinement and the optimal, bottom-up simplification scheme. The graph shows for two different object space metrics the minimal number of triangles (shaded areas) and relative overhead (solid lines) due to refinement. The data was gathered over 2,816 views for an error tolerance of $\tau = 1$ pixel.

the relative increase in number of triangles from using top-down refinement versus bottom-up simplification. Notice that, when the overall triangle counts are high (frames 1024–2048), the relative increase is on the order of 1-2% for both metrics. The overhead becomes large only when the triangle counts are low, suggesting that our refinement produces a small, roughly constant increase in number of triangles. However, because the large peaks occur only at low triangle counts, the net increase in number of triangles remains low. Over the entire fly-over, the total number of rendered triangles increased only by 1.63% and 3.76% for the maximum and incremental errors, respectively. These averages appear in the histograms in Fig. 16, which show the highly skewed distributions of the mesh complexity overhead for the two metrics. Because $\hat{\epsilon}^{max}$ is already close to nested, using simplification instead of refinement has less potential for improvement than in the case of incremental errors.

For the same flight path as above, over the full-resolution data set, we also measured the mesh complexity for the isotropic and anisotropic screen space error metrics, using $\hat{\epsilon}^{inc}$ as the object space metric. We found that the anisotropic metric on average led to a 2.5% reduction in mesh complexity over the isotropic metric, although at the expense of efficiency of evaluation. This

[6] As noted earlier, it is possible that removing a vertex leads to a reduction in $\hat{\epsilon}$ (e.g. resulting in a mesh with a better fit), and thus possibly a reduction in ρ from above to below τ. Whether such a coarsening operation should be allowed or not depends on the interpretation of the error metric. This choice, however, clearly has an impact on what constitutes a "minimal mesh." For consistency and simplicity, we have chosen to be conservative and not allow such operations.



(a) Incremental error.

(b) Maximum error.

Fig. 16. Distribution and average (dashed line) of overhead in mesh complexity due to conservative refinement relative to the minimal mesh.

platform	*geo-morphing*	*multi-threading*	*view culling*	*time (s)*	*rendering (Mtri/s)*	*refinement (Mtri/s)*
15.5 GB SGI				317.43	0.939	1.435
			✓	75.50	0.595	1.466
		✓	✓	47.63	0.944	1.396
	✓	✓	✓	47.71	0.942	0.990
900 MB PC				170.70	1.683	2.350
			✓	43.89	0.980	1.860
		✓	✓	38.62	1.113	1.777
	✓	✓	✓	38.98	1.103	1.515

TABLE VII

FLIGHT TIME AND AVERAGE PERFORMANCE FOR 2816-FRAME FLY-OVER (SEE ALSO FIG. 17). THE RENDERING PERFORMANCE IS MEASURED AS THE NUMBER OF RENDERED TRIANGLES OVER THE FRAME TIME (IN MULTIPLES OF THE MONITOR FRAME TIME), WHICH INCLUDES THE REFINEMENT TIME IN SINGLE-THREADED MODE.

behavior, which was also observed by Hoppe [8], leads us to conclude that the simple isotropic metric is to be favored.

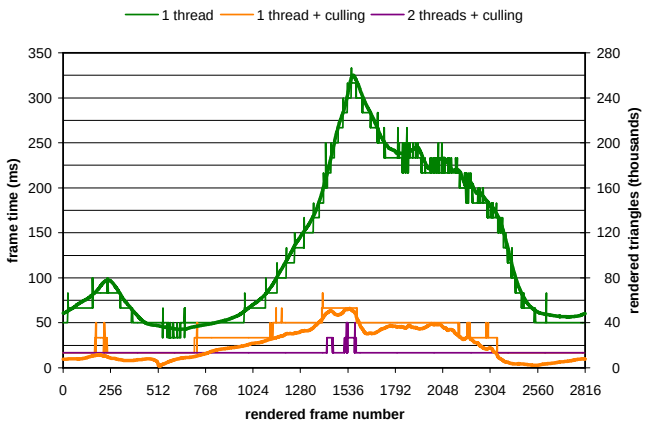
We next evaluate the performance increase due to the use of culling and multi-threading (one thread each for rendering and refinement). These results, which are summarized in Table VII and plotted in Fig. 17, demonstrate a clear advantage of using both culling and multi-threading. We were able to sustain up to 40,000 rendered triangles at 60 frames per second on the SGI Onyx2 during the fly-over. When the number of rendered triangles exceeded 40,000, however, the frame rate slowed briefly. We generally obtained even higher frame rates on the PC (over 72 Hz on average), but the rates were more varied. In both cases, we synchronized our rendering rate with the monitor display rate (60 Hz on the SGI, 75 Hz on the PC), which in many cases resulted in significant idle time. This idle time is part of the overall frame time used to measure the rendering speed in Table VII.

Fig. 17(b) highlights the refinement performance, with and without culling, measured in number of rendered triangles divided by the wall clock refinement time. For low triangle counts, the refinement runs faster when view culling is disabled, as expected. Notice, however, that as the mesh complexity increases towards the middle of the graph, the lack of view culling leads to a significant decrease in performance. Conversely, the use of view culling results in a relative speedup. We attribute this result to caching behavior—as the triangle strip grows, an increasing number of cache misses are made, which slows down the method that did not use culling. Meanwhile, when a large fraction of triangles are culled, the overhead of making recursive function calls dominates, as evidenced by the sharp drop in performance near frame #512.

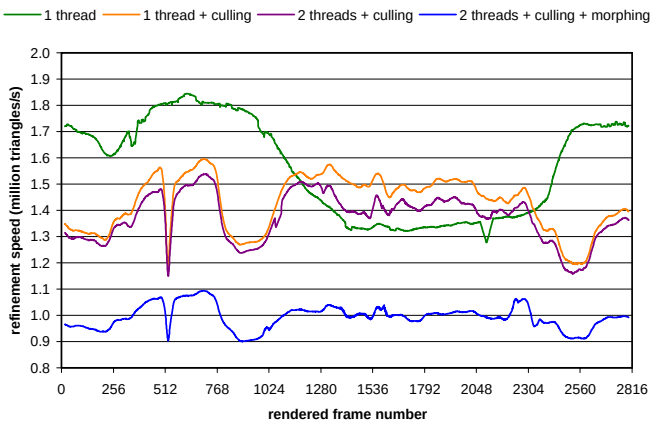
In order to show the qualitative performance of our geomorphs, we have prepared a number of MPEG animations that can be downloadeded from http://www.cc.gatech.edu/~lindstro/papers/tvcg2002/. These animations indicate that by using geomorphing the screen space error tolerance can be increased considerably without any appreciable loss in rendered quality. Objectionable temporal artifact like geometric popping and shading discontinuities are virtually eliminated by smoothly morphing the geometry. Without geomorphing, popping artifacts can be visually disturbing if the tolerance is larger than two pixels. Using geomorphing, the lower error tolerance τ_{min} can be doubled or even tripled before the smooth motion caused by the geomorphs can even be detected.

Table VII and Fig. 17(b) show some quantitative results of using geomorphing. Using $\tau_{max} = \frac{3}{2}\tau_{min}$ and a variety of choices for τ_{min}, we found that roughly 20–25% of the vertices were morphing at any one time. As expected, the additional work required to continuously morph the mesh geometry results in a drop in the refinement performance. However, as our animations show, the increase in acceptable tolerance due to improvements in temporal quality more than offsets the comparatively small increase in refinement time.

Finally, we evaluated the efficiency of using a single triangle strip. We found that the ratio of triangle strip vertices to the number of non-degenerate triangles averaged 1.56 vertices/triangle with virtually no variance. This number should be compared to 3 vertices/triangle for a list of independent triangles.



(a) Frame time (thin lines) and number of rendered triangles (thick lines).



(b) Refinement performance over time.

Fig. 17. In-core rendering and refinement performance on SGI Onyx2 over several thousand frames of the Puget Sound fly-over. The curves correspond to the use of single-threading—with and without culling—and multi-threading with culling—with and without geomorphing. The hierarchical indexing scheme was used in all four runs. (a) Using multi-threading a steady 60 Hz is maintained during nearly the entire fly-over. The number of triangles for the three schemes that used culling coincide, therefore the graph for only one of them is shown. Similarly, the frame rates with and without geomorphing were roughly the same. (b) The vertical axis corresponds to the number of non-degenerate triangles in the triangle strip divided by the (wall clock) refinement/culling/geomorphing time.

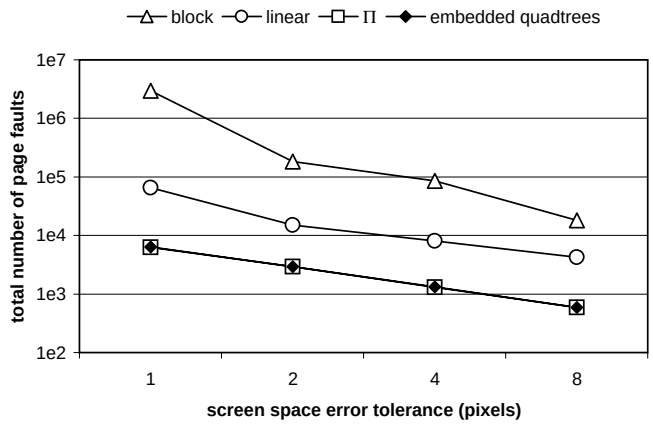


Fig. 18. Total number of page faults vs. screen space error tolerance τ on 900 MB SGI.

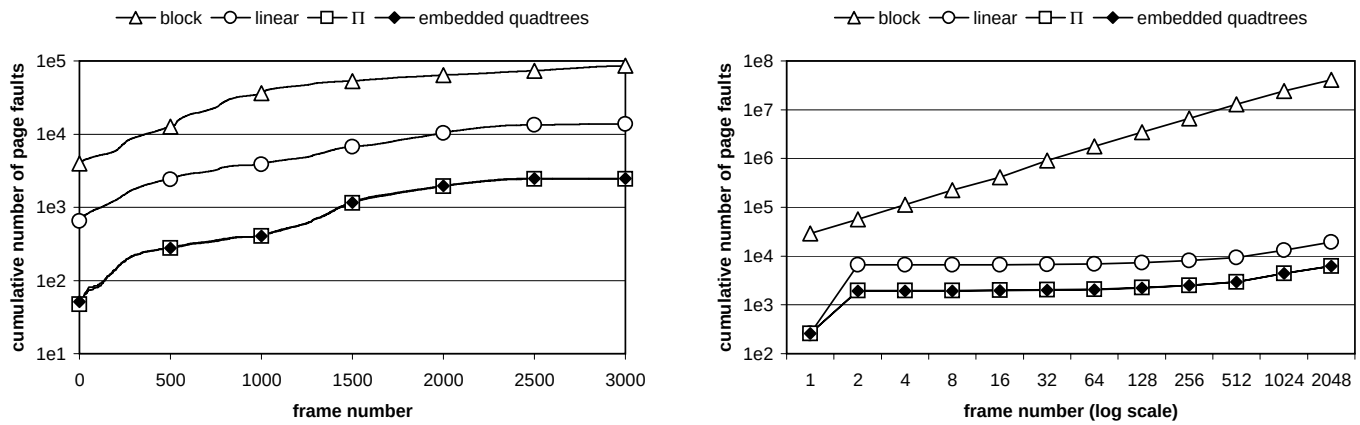
B. Data Layout

In this section, we compare the memory performance of four different indexing schemes: the single quadtree scheme from Section IV-B, where the "white" tree is embedded in the "black" tree; the Π-order indexing scheme; a blocking scheme based on 32×32 tiles from the highest resolution data; and a standard matrix layout in row major form. For all these methods, we stored the fields $\langle \mathbf{p}, \epsilon, r \rangle$, which together occupy 20 bytes, with each vertex (see Section III). Our focus here is not on the storage efficiency of the vertex records—it is entirely possible to compress or even eliminate some fields in this record (see Section VII-B). Rather, we assume fixed-length records and focus on how efficient the different indexing schemes are at accessing them.

Fig. 18 shows for the Puget Sound fly-over the total number of page faults for varying values of the error tolerance τ. Smaller values of τ result in larger meshes being rendered and more data being paged in. Clearly, the hierarchical indexing schemes (quadtree-based and Π-order) greatly outperformed the linear and block-based schemes, and often lead to drastically improved paging speeds (Fig. 20). Perhaps surprisingly, the block-based scheme, which is often used for terrains, performs the worst of them all. This is because the refined mesh rarely consists of groups of many vertices at the highest resolution. Instead, a handful of vertices are needed from each block, requiring virtually the entire terrain to be paged in during each refinement pass. A more reasonable block-based indexing scheme would be to subsample the data and create a multiresolution pyramid, allowing more coherent access to different resolutions of data. However, such an indexing scheme uses multiple indices for each vertex, which would arguably make for an unfair comparison with our other indexing schemes.

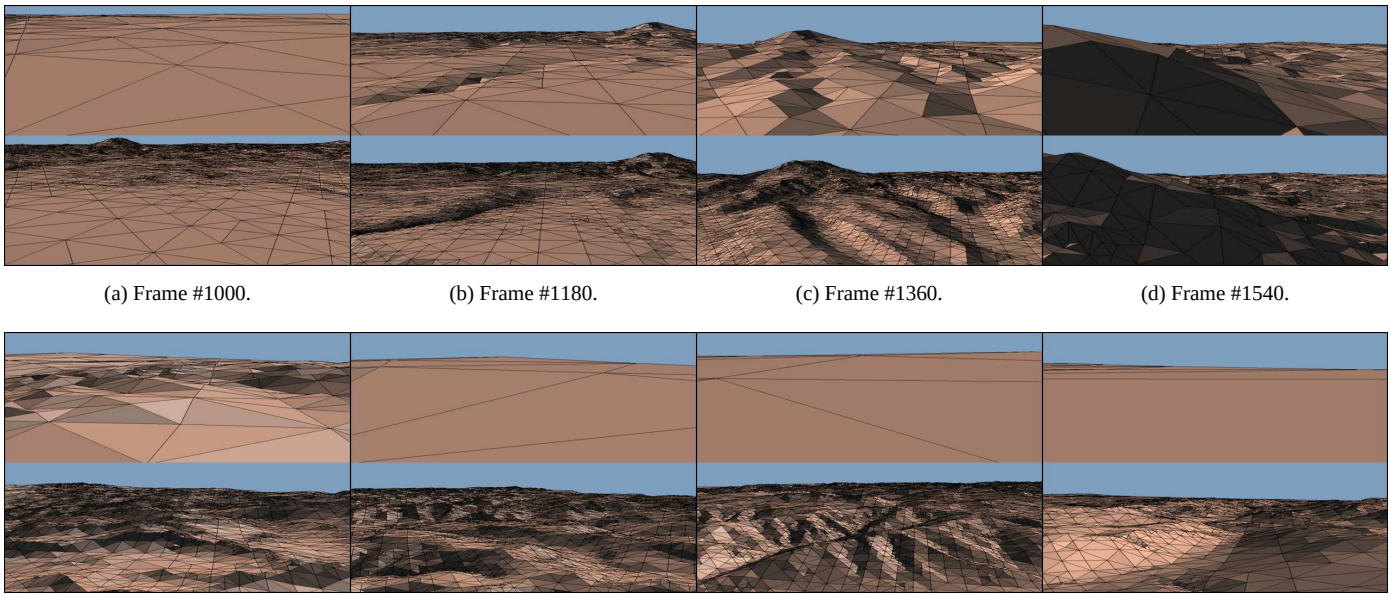
We also investigated the paging behavior over time. Results for the SGI Octane are shown in Fig. 19(a), while the PC results are shown (on a log-log scale) in Fig. 19(b). These graphs show that there is a significant hit at startup, when no data is memory resident, after which pages often stay in user memory or can be reclaimed quickly from the operating system's cache. These results also indicate that the hierarchical indexing schemes consistently result in one to two orders of magnitude improvement in paging performance over the non-hierarchical layouts.

Finally, we measured the raw in-core refinement speed of all indexing schemes. Due to better cache locality, the quadtree scheme, while involving a few more operations, is still twice as fast as the linear scheme, and is also twice as fast as the more complex Π-order scheme. This suggests that the linear scheme is inferior in all aspects to quadtree-based indexing, with the exception of memory overhead. We plan to investigate alternative indexing schemes that have the same desirable properties as the quadtree scheme, but with higher memory efficiency.



(a) 900 MB SGI. (b) 64 MB PC

Fig. 19. Cumulative number of page faults over time on two different platforms.



(a) Frame #1000. (b) Frame #1180. (c) Frame #1360. (d) Frame #1540.

(e) Frame #1720. (f) Frame #1900. (g) Frame #2080. (h) Frame #2260.

Fig. 20. Frames taken every three seconds from two multi-threaded fly-over sequences using linear, row major indexing (top half of each frame) and quadtree-based indexing (bottom half). The flight paths for the two sequences are the same. The improved cache performance of the quadtree-based scheme results in more detail being paged in more quickly.

VII. DISCUSSION

At this point, we would like to discuss ongoing research and other topics relevant to our visualization algorithms, as well as directions for future work.

A. General Terrain Data

The height fields representable by edge bisection are necessarily restricted to be of dimensions $2^n + 1$ squared. For height fields that do not fit these dimensions, we currently expand them to the next larger square power of two. Because we store the full xyz-coordinates of each vertex, we can preserve the areal extent of the height field by clamping the coordinates of the vertices that fall outside the input region and initializing their errors and radii to zero. Perhaps a better solution to this problem is to partition the terrain up into smaller square blocks, while propagating the error and radius information between blocks during preprocessing to ensure that no cracks between them are created.

Another constraint in our current system is that the input data be a single grid sampled at a uniform resolution. In many applications, multiple, possibly nested data sets at varying resolution need to be georeferenced and integrated into a single data set [23, 24]. Using our current approach, we would need to resample all data sets to a common highest resolution, which is impractical. Instead, we suggest partitioning the terrain both spatially and in resolution using a meta-hierarchy of blocks, such as a quadtree, to organize the data. Our algorithms would then be used in their current form within each block in the meta-hierarchy.

Not all height fields are represented as regular grids. In fact,

more general representations, such as TINs, exist for a good reason; the amount of spatial complexity often varies over a surface. While our adaptive refinement capitalizes on this fact, the source data is still stored as a uniform grid. In addition, because the mesh is made up of a fixed pattern of right isosceles triangles, preserving sharp geometric features, such as ridge lines, and image features, such as roads and rivers, is difficult. As demonstrated in [47], there is no need for the refinement to insert vertices at edge midpoints. Using our current data structures, which store the xyz-coordinates with each vertex, we could perform a data-dependent triangulation that still has the same subdivision connectivity. The key research problem lies in how to efficiently and optimally construct a good triangulation with subdivision connectivity.

B. Compression

Based on the data structures discussed in Section V, we require twenty bytes of storage per vertex. Using the quadtree layouts from Section IV, this number effectively increases by one or two thirds. While our memory and disk footprints are not nearly as large as some competing methods, e.g. [8], storage efficiency is still a concern. If space is at a premium, it is possible to compress the per-vertex information considerably. Without going into great detail, we here sketch a possible scheme for encoding each vertex using only 16 bits (assuming the original height field can be represented using 16 bits).

As already mentioned, the xy-coordinates of each vertex can be computed on-the-fly. Thus we are concerned only with encoding $\langle z, \epsilon, r \rangle$. Because most height fields are relatively smooth, we expect the elevation to be highly correlated at all scales. That is, we can use the existing hierarchy as a linear predictor for z by using the midpoint of the associated split edge as the estimate. We would then store only the signed difference δ from the estimate. This is in a sense similar to a linear wavelet transform of the data. We would expect the magnitude of δ on average to be much smaller than z, and therefore require fewer bits to store.

Note that for any reasonable object space metric, we have $\epsilon \geq |\delta|$, i.e. the error ought to be at least as large as the deviation between the mesh before and after removing the vertex. Also, we know that ϵ decreases monotonically with each refinement level. Based on these facts, a variant of zerotree coding [53] can be used to progressively eliminate redundant bits. That is, if we know that ϵ_i for vertex i is smaller than some threshold, then $|\delta_i|$ must also be smaller, as are ϵ_j and $|\delta_j|$ for all descendants j of i. In particular, if the most significant bit (or bits) of ϵ_i is zero, then we do not have to encode this bit (bits) for either ϵ_j or $|\delta_j|$ on the remaining levels. Thus, as ϵ grows smaller level by level, progressively fewer bits are needed to represent ϵ and δ, and more bits become available for encoding r. Note that when $|\delta|$ is large, we already expect the vertex to be active, and representing ϵ and r less accurately may in this case be acceptable. The only caveat here is that the number of significant bits used to encode ϵ and δ for a particular vertex must be independent of how we reached the vertex. Thus, both DAG parents of a vertex must agree on how many bits to use for their common child.

To ensure that the height field is represented losslessly (to the given 16-bit precision), we suggest encoding all significant bits of δ and using any remaining bits for a lossy, conservative encoding of ϵ and r, i.e. the least significant bits that are not encoded are all assumed to be ones. To encode the bounding sphere radius r, we note that r is highly correlated on any given level, and is bounded below by the smallest radius needed for nesting circles in 2D. Thus, we can encode the excess of the radii using small per-level lookup tables.

For efficiency reasons, we must avoid having to uncompress the vertex information every time a vertex is touched. Therefore, we advocate using a fast caching mechanism, similar to the low-level caches used in modern CPUs, to reuse previously uncompressed vertices. The size of this cache should be chosen to be roughly proportional to the size of the largest adaptive mesh.

Whereas the compression scheme described above reduces the storage requirements by a factor of ten, the largest potential for compression comes from using variable-length coding, which would allow large regions of the height field to be compressed using only a few bits. For example, large subtrees of unused ghost vertices (Section IV-A) could be eliminated this way. Similarly, for composite data sets that vary in resolution, and for data sets that do not fit the size requirements imposed by hierarchical bisection, large unused portions of the vertex data could virtually be eliminated using variable-length compression.

VIII. SUMMARY

We have presented algorithms for two important components of large-scale terrain rendering: a method for efficient view-dependent refinement; and an indexing scheme for organizing the data in a memory-friendly manner. The refinement and rendering components of our algorithm have been shown to be very efficient, and in spite of their simplicity compete with the state of the art in terrain visualization. Indeed, the core components of our view-dependent refinement and hierarchical indexing can be implemented in as little as a few dozen lines of C code. An implementation of our algorithms can be downloaded from http://www.cc.gatech.edu/~lindstro/software/soar/.

We have already discussed some possible directions for future work, but briefly mention a few additional ideas. Whereas the majority of recent work on terrain visualization has been on view-dependent geometric approximation, perhaps an even more important component is texturing and texture level-of-detail management. Few techniques currently exist for transparently caching and loading textures. To our knowledge, the only general approach to scalable texture caching is the SGI-specific extension for *clip mapping* [54]. Having a general paging mechanism such as memory mapping for hierarchical textures would be tremendously useful. Even in the case of explicit texture paging, we see a lot of room for improvement. A related problem is how to perform efficient high-quality shading, which is often implemented by pre-shading the geometry and storing the result in a high-resolution texture. Achieving high-quality dynamic lighting is still a largely unresolved problem. Another issue related to view-dependent methods, and in particular those that incorporate geomorphing, is how to efficiently render the continuously changing mesh. Current graphics hardware is not optimized for "immediate mode" rendering of dynamic meshes, and thus hybrid methods may be neeed that cache larger, static

pieces of the mesh for more efficient rendering. Finally, we envision that our algorithms for 2D height fields can be generalized to higher dimensions. For example, we intend to investigate how to extend our framework to view-dependent rendering of 3D scalar fields using techniques such as progressive isocontouring.

ACKNOWLEDGEMENTS

This work was performed under the auspices of the U.S. DOE by LLNL under contract no. W-7405-Eng-48.

REFERENCES

[1] Peter Lindstrom and Valerio Pascucci, "Visualization of large terrains made easy," in *IEEE Visualization 2001*, Kenneth I. Joy, Amitabh Varshney, and Thomas Ertl, Eds., San Diego, California, Oct. 2001, pp. 363–370, IEEE.

[2] Peter Lindstrom, David Koller, William Ribarsky, Larry F. Hodges, Nick Faust, and Gregory Turner, "Real-time, continuous level of detail rendering of height fields," in *Proceedings of SIGGRAPH 96*, Holly Rushmeier, Ed., New Orleans, Louisiana, Aug. 1996, Computer Graphics Proceedings, Annual Conference Series, pp. 109–118, Addison Wesley.

[3] Mark A. Duchaineau, Murray Wolinsky, David E. Sigeti, Mark C. Miller, Charles Aldrich, and Mark B. Mineev-Weinstein, "ROAMing terrain: Real-time optimally adapting meshes," in *IEEE Visualization '97*, Roni Yagel and Hans Hagen, Eds., Phoenix, Arizona, Nov. 1997, pp. 81–88, IEEE.

[4] Renato B. Pajarola, "Large scale terrain visualization using the restricted quadtree triangulation," in *IEEE Visualization '98*, David Ebert, Hans Hagen, and Holly Rushmeier, Eds., Research Triangle Park, North Carolina, Oct. 1998, pp. 19–26, IEEE.

[5] William Evans, David Kirkpatrick, and Gregg Townsend, "Right-triangulated irregular networks," *Algorithmica*, vol. 30, no. 2, pp. 264–286, Mar. 2001.

[6] Hugues Hoppe, "View-dependent refinement of progressive meshes," in *Proceedings of SIGGRAPH 97*, Turner Whitted, Ed., Los Angeles, California, Aug. 1997, Computer Graphics Proceedings, Annual Conference Series, pp. 189–198, ACM Press.

[7] Markus Gross, Roger Gatti, and Oliver Staadt, "Fast multiresolution surface meshing," in *IEEE Visualization '95*, Gregory M. Nielson and Deborah Silver, Eds., Atlanta, Georgia, Oct. 1995, pp. 135–142, IEEE Computer Society Press.

[8] Hugues Hoppe, "Smooth view-dependent level-of-detail control and its application to terrain rendering," in *IEEE Visualization '98*, David Ebert, Hans Hagen, and Holly Rushmeier, Eds., Research Triangle Park, North Carolina, Oct. 1998, pp. 35–42, IEEE.

[9] Stefan Röttger, Wolfgang Heidrich, Philipp Slussallek, and Hans-Peter Seidel, "Real-time generation of continuous levels of detail for height fields," in *Proceedings of the 6th International Conference in Central Europe on Computer Graphics and Visualization*, Feb. 1998, pp. 315–322.

[10] Jonathan Blow, "Terrain rendering at high levels of detail," in *Proceedings of the 2000 Game Developers Conference*, San Jose, California, Mar. 2000.

[11] Thomas Gerstner, "Multiresolution visualization and compression of global topographic data," *Geoinformatica*, 2002, To appear. Available as SFB 256 Report 29, University of Bonn, 1999.

[12] Robert L. Ferguson, Richard Economy, William A. Kelly, and Pedro P. Ramos, "Continuous terrain level of detail for visual simulation," in *Proceedings of the 1990 IMAGE V Conference*, Tempe, Arizona, June 1990, pp. 144–151.

[13] Daniel Cohen-Or and Yishay Levanoni, "Temporal continuity of levels of detail in delaunay triangulated terrain," in *IEEE Visualization '96*, Roni Yagel and Gregory M. Nielson, Eds., San Francisco, California, Oct. 1996, pp. 37–42, IEEE.

[14] John Rohlf and James Helman, "IRIS Performer: A high performance multiprocessing toolkit for real-time 3d graphics," in *Proceedings of SIGGRAPH 94*, Andrew Glassner, Ed., Orlando, Florida, July 1994, Computer Graphics Proceedings, Annual Conference Series, pp. 381–395, ACM Press.

[15] David Cline and Parris K. Egbert, "Terrain decimation through quadtree morphing," *IEEE Transactions on Visualization and Computer Graphics*, vol. 7, no. 1, pp. 62–69, Jan. - Mar. 2001.

[16] Jeffrey S. Vitter, "External memory algorithms and data structures: Dealing with MASSIVE DATA," *ACM Computing Surveys*, vol. 33, no. 2, pp. 209–271, 2001.

[17] Y. Matias, E. Segal, and J. S. Vitter, "Efficient bundle sorting," in *Proceedings of the 11th Annual SIAM/ACM Symposium on Discrete Algorithms*, San Francisco, California, Jan. 2000, pp. 839–848.

[18] Lars Arge and Peter Bro Miltersen, "On showing lower bounds for external-memory computational geometry problems," in *External Memory Algorithms and Visualization*, James Abello and Jeffrey Scott Vitter, Eds., DIMACS Series in Discrete Mathematics and Theoretical Computer Science. American Mathematical Society Press, Providence, Rhode Island, 1999.

[19] M. T. Goodrich, J.-J. Tsay, D. E. Vengroff, and J. S. Vitter, "External-memory computational geometry," in *Proceedings of the 34th Annual IEEE Symposium on Foundations of Computer Science*, Palo Alto, California, Nov. 1993.

[20] Yi-Jen Chiang and Cláudio T. Silva, "I/O optimal isosurface extraction," in *IEEE Visualization '97*, Roni Yagel and Hans Hagen, Eds., Phoenix, Arizona, Nov. 1997, pp. 293–300, IEEE.

[21] C. L. Bajaj, V. Pascucci, D. Thompson, and X. Y. Zhang, "Parallel accelerated isocontouring for out-of-core visualization," in *Proceedings of the 1999 IEEE Parallel Visualization and Graphics Symposium*, Stephan N. Spencer, Ed. Oct. 1999, pp. 97–104, ACM SIGGRAPH.

[22] Douglass Davis, T. Y. Jiang, William Ribarsky, and Nickolas Faust, "Intent, perception, and out-of-core visualization applied to terrain," in *IEEE Visualization '98*, David Ebert, Hans Hagen, and Holly Rushmeier, Eds., Research Triangle Park, North Carolina, Oct. 1998, pp. 455–458, IEEE.

[23] Peter Lindstrom, David Koller, William Ribarsky, Larry F. Hodges, Augusto Op den Bosch, and Nick Faust, "An integrated global gis and visual simulation system," Tech. Rep. GIT-GVU-97-07, Georgia Institute of Technology, Mar. 1997.

[24] Martin Reddy, Yvan Leclerc, Lee Iverson, and Nat Bletter, "Terravision ii: Visualizing massive terrain databases in VRML," *IEEE Computer Graphics & Applications*, vol. 19, no. 2, pp. 30–38, Mar. - Apr. 1999.

[25] Craig Gotsman and Boris Rabinovitch, "Visualization of large terrains in resource-limited computing environments," in *IEEE Visualization '97*, Roni Yagel and Hans Hagen, Eds., Phoenix, Arizona, Nov. 1997, pp. 95–102, IEEE.

[26] Dru Clark and Michael Bailey, "Visualization of height field data with physical models and texture photomapping," in *IEEE Visualization '97*, Roni Yagel and Hans Hagen, Eds., Phoenix, Arizona, Nov. 1997, pp. 89–94, IEEE.

[27] Jürgen Döllner, Konstantin Baumann, and Klaus Hinrichs, "Texturing techniques for terrain visualization," in *IEEE Visualization 2000*, Thomas Ertl, Bernd Hamann, and Amitabh Varshney, Eds., Salt Lake City, Utah, Oct. 2000, pp. 207–234, IEEE.

[28] Hans Sagan, *Space-Filling Curves*, Springer-Verlag, New York, New York, 1994.

[29] M. Parashar, J.C. Browne, C. Edwards, and K. Klimkowski, "A common data management infrastructure for adaptive algorithms for PDE solutions," in *Proceedings of Supercomputing '97*, San Jose, California, Nov. 1997, ACM SIGARCH and IEEE.

[30] M. Griebel and G. W. Zumbusch, "Parallel multigrid in an adaptive PDE solver based on hashing and space-filling curves," *Parallel Computing*, vol. 25, no. 7, pp. 827–843, July 1999.

[31] R. Niedermeier, K. Reinhardt, and P. Sanders, "Towards optimal locality in mesh-indexings," in *Proceedings of Fundamentals of Computation Theory '97*. Sept. 1997, vol. 1279, pp. 364–375, Springer-Verlag.

[32] D. Hilbert, "Über die stetige abbildung einer linie auf ein flachenstück," *Mathematische Annalen*, vol. 38, pp. 459–460, 1891.

[33] Rolf Niedermeier and Peter Sanders, "On the manhattan-distance between points on space-filling mesh-indexings," Tech. Rep. 1996-18, Universität Karlsruhe, Informatik für Ingenieure und Naturwissenschaftler, 1996.

[34] S.-I. Kamata and Y. Bandou, "An address generator of a pseudo-Hilbert scan in a rectangle region," in *International Conference on Image Processing, ICIP 97*, 1997, pp. 707–714.

[35] Y. Bandou and S.-I. Kamata, "An address generator for an n-dimensional pseudo-Hilbert scan in a hyper-rectangular parallelepiped region," in *International Conference on Image Processing, ICIP 2000*, 2000, pp. 707–714.

[36] J. K. Lawder, *The Application of Space-filling Curves to the Storage and Retrieval of Multi-Dimensional Data*, Ph.D. thesis, School of Computer Science and Information Systems, Birkbeck College, University of London, 2000.

[37] Laurent Balmelli, *Rate-Distortion Optimal Mesh Simplification for Communications*, Ph.D. thesis, Ecole Polytechnique Federale, Switzerland, 2001.

[38] Hanan Samet, *Applications of Spatial Data Structures*, Addison-Wesley, Reading, Massachusetts, 1990.

[39] Siddhartha Chatterjee, Alvin R. Lebeck, Praveen K. Patnala, and Mithuna Thottethodi, "Recursive array layouts and fast parallel matrix multiplication," in *Proceedings of the 11th Annual ACM Symposium on*

Parallel Algorithms and Architectures, Saint-Malo, France, June 1999, SIGACT/SIGARCH and EATCS, pp. 222–231.

[40] Jeremy D. Frens and David S. Wise, "Auto-blocking matrix-multiplication or tracking BLAS3 performance from source code," *ACM SIGPLAN Notices*, vol. 32, no. 7, pp. 206–216, July 1997.

[41] David S. Wise, "Ahnentafel indexing into Morton-ordered arrays, or matrix locality for free," in *Proceedings of EUROPAR 2000: Parallel Processing*. Aug. 2000, vol. 1900 of *Lecture Notes in Computer Science*, pp. 774–784, Springer.

[42] Valerio Pascucci, "Multi-resolution indexing for hierarchical out-of-core traversal of rectilinear grids," in *NSF/DoE Lake Tahoe Workshop on Hierarchical Approximation and Geometrical Methods for Scientific Visualization*, Tahoe City, California, Oct. 2000, Online proceedings at http://muldoon.cipic.ucdavis.edu/~hvm00/program.html. LLNL Technical report UCRL-JC-140581.

[43] Valerio Pascucci and Randall J. Frank, "Global static indexing for real-time exploration of very large regular grids," in *Proceedings of Supercomputing 2001*, Denver, Colorado, Nov. 2001, Available as LLNL technical report UCRL-JC-144754.

[44] Mark de Berg and Katrin T. G. Dobrint, "On levels of detail in terrains," in *Proceedings of ACM Symposium on Computational Geometry*, Vancouver, British Columbia, June 1995, pp. C26–C27.

[45] Michael Garland and Paul Heckbert, "Fast polygonal approximation of terrains and height fields," Tech. Rep. CMU-CS-95-181, Carnegie Mellon University, Sept. 1995.

[46] Leila De Floriani, Paola Magillo, and Enrico Puppo, "Efficient implementation of multi-triangulations," in *IEEE Visualization '98*, David Ebert, Hans Hagen, and Holly Rushmeier, Eds., Research Triangle Park, North Carolina, Oct. 1998, pp. 43–50, IEEE.

[47] Luiz Velho and Jonas Gomes, "Variable resolution 4-k meshes: Concepts and applications," *Computer Graphics Forum*, vol. 19, no. 4, pp. 195–212, Dec. 2000.

[48] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest, *Introduction to Algorithms*, MIT Press, 1990.

[49] Thomas Gerstner, Martin Rumpf, and Ulrich Weikard, "Error indicators for multilevel visualization and computing on nested grids," *Computers & Graphics*, vol. 24, no. 3, pp. 363–373, June 2000.

[50] Francine Evans, Steven S. Skiena, and Amitabh Varshney, "Optimizing triangle strips for fast rendering," in *IEEE Visualization '96*, Roni Yagel and Gregory M. Nielson, Eds., San Francisco, California, Oct. 1996, pp. 319–326, IEEE.

[51] Szymon Rusinkiewicz and Marc Levoy, "QSplat: A multiresolution point rendering system for large meshes," in *Proceedings of SIGGRAPH 2000*, Kurt Akeley, Ed., New Orleans, Louisiana, July 2000, Computer Graphics Proceedings, Annual Conference Series, pp. 343–352, ACM Press / ACM SIGGRAPH / Addison Wesley Longman.

[52] Lee R. Willis, Michael T. Jones, and Jenny Zhao, "A method for continuous adaptive terrain," in *Proceedings of the 1996 Image Conference*, Scottsdale, Arizona, June 1996, pp. 23–28.

[53] Jerome M. Shapiro, "Embedded image coding using zerotrees of wavelet coefficients," *IEEE Transactions on Signal Processing*, vol. 41, no. 12, pp. 3445–3462, Dec. 1993.

[54] Christopher C. Tanner, Christopher J. Migdal, and Michael T. Jones, "The Clipmap: A virtual mipmap," in *Proceedings of SIGGRAPH 98*, Michael Cohen, Ed., Orlando, Florida, July 1998, Computer Graphics Proceedings, Annual Conference Series, pp. 151–158, ACM Press.

APPENDIX

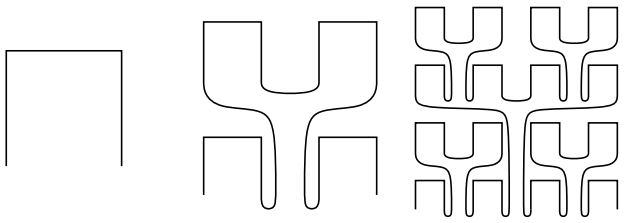
I. Π-ORDER SPACE FILLING CURVE DATA LAYOUT.

In this appendix we construct an alternative data layout based on the Π-order space-filing curve (a variation of the Z-order Peano curve [28]). Contrary to the quadtree indexing schemes in Section IV, this data layout does not require extraneous ghost vertices, and therefore does not suffer from the 33% storage overhead associated with those schemes. Unfortunately, the gain in storage efficiency comes at the expense of a more complicated implementation. Our approach is based upon, and relies heavily on the related hierarchical Z-order indexing scheme described in [42, 43], which applies neatly to grid dimensions of 2^n.[7] We here derive an extension of this scheme to grids with $2^n + 1$ vertices by treating the extra row and column as a special boundary case. For background information and a more detailed exposition, we refer the reader to [42, 43]. For the sake of completeness, we here provide only the details necessary for implementing the Π-order scheme.

We will first derive a "flat" index for a single-resolution grid. Then, in Section A-B, we describe how to extend it to a hierarchical layout, such that all vertices on a given level have indices smaller than the vertices on the next finer refinement level.

A. Local Index

Fig. 21 shows the recursive construction of the Π-order space filling curve. The basic Π pattern involving four vertices is shown in Fig. 21(a). One level of resolution is built by replacing each vertex on the previous level with a Π shape reduced in scale by a factor of two. The space filling curve built over n levels defines a total order on the vertices of a $2^n \times 2^n$ grid. Fig. 21(c) shows this total order for a grid of 8×8 vertices.

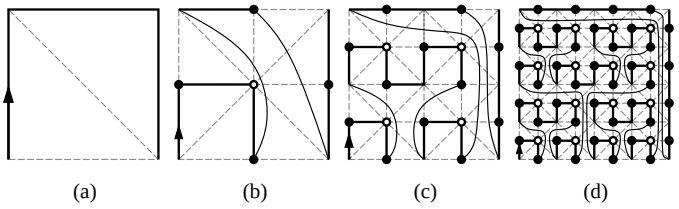


(a) (b) (c)

Fig. 21. Recursive construction of the Π-order space filling curve. (a) Basic level formed by a simple Π pattern. (b) Second level. Each vertex in the basic level is replaced by a Π pattern scaled by 1/2. (c) Third level. Each vertex of the second level is replaced by a Π pattern scaled by 1/4.

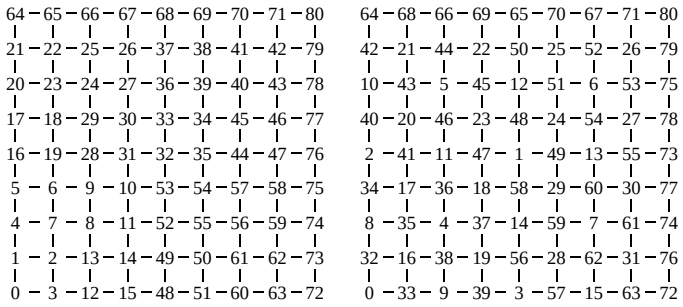
To make direct use of this data layout we need to align it to a grid of size $(2^n + 1) \times (2^n + 1)$. Moreover, the hierarchy of the longest edge bisection refinement must be matched with the recursive structure of the space filling curve. Fig. 22 shows the recursive construction of this data layout for a 9×9 grid ($n = 3$). The vertices of the base mesh (a) are traversed in clockwise order, starting from the bottom left corner. In each refinement every interior (i.e. non-boundary) vertex is replaced with a Π pattern that includes one new white vertex (first bisection) and two new black vertices (second bisection). The refinement of boundary vertices, on the other hand, results in the creation of a single black vertex. At the end of this construction, each vertex v of the grid is associated with a unique index i in the range $0 \le i \le (2^{2n} + 2^{n+1}) = (2^n + 1)^2 - 1$.

[7] We favor the Π layout over the Z layout because the hierarchical Π-order index is a better match with the order in which vertices are visited during mesh refinement.



(a) (b) (c) (d)

Fig. 22. Π-order curve with boundary (solid black lines) aligned with the vertices of a rectilinear grid (dashed gray lines). Each new level in the Π-order curve corresponds to two levels of edge bisection refinement of the grid. The first bisection introduces the white vertices while the second bisection introduces the black vertices (see Fig. 2). The vertices in the top row and in the right column constitute the added boundary that allows a grid of $(2^n + 1) \times (2^n + 1)$ vertices to be covered exactly.

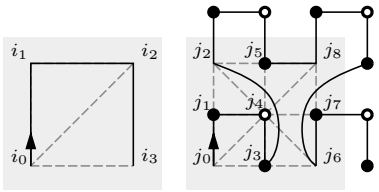


(a) (b)

Fig. 23. Indices of the vertices of a 9×9 grid. (a) Basic Π-order with boundary. (b) Hierarchical version of the same index.

Fig. 23(a) shows the index of all the vertices in a 9×9 grid. During the hierarchical traversal, this index is computed by making simple bitwise modifications to the indices from the immediately coarser level. We use a function index-append$(i, k, l) = i + 2^l k, k \in \{1, 2, 3\}$, which sets one or two bits of i. Here k corresponds to one of the three "children" in the pi-order refinement, that is above ($k = 1$), above-right ($k = 2$), or right of ($k = 3$) i. As in Section IV we assume that the vertices at the finest refinement resolution are on level $l = 1$, while coarser levels have increasingly higher values of l. We focus first on the refinement of a single square in the interior of the grid. Fig. 24(a) shows this configuration. The coarse vertices of the square are represented by the indices i_0, i_1, i_2, and i_3. The grid of nine vertices obtained after the refinement is represented by the indices $j_0, \ldots, j_8$. First of all the indices of the four corners do not change: $j_0 \leftarrow i_0$, $j_2 \leftarrow i_1$, $j_6 \leftarrow i_3$, $j_8 \leftarrow i_2$. The indices j_1, j_3 and j_4 are computed from i_0 as follows:

$$j_1 \leftarrow \text{index-append}(i_0, 1, l-2)$$
$$j_3 \leftarrow \text{index-append}(i_0, 3, l-2)$$
$$j_4 \leftarrow \text{index-append}(i_0, 2, l-2)$$



(a) (b)

Fig. 24. One step of refinement of the Π-order curve in the interior of the grid (no boundary vertices are present). The shaded regions correspond to the same area. (a) Vertices before refinement. (b) Vertices after refinement. Note that $i_0 = j_0$, $i_1 = j_2$, $i_2 = j_8$, and $i_3 = j_6$.

Similarly, j_5 and j_7 are computed from i_1 and i_3 as follows:

$$j_5 \leftarrow \textsf{index-append}(i_1, 3, l-2)$$
$$j_7 \leftarrow \textsf{index-append}(i_3, 1, l-2)$$

Exceptions to these rules are the boundary cases, i.e. where either i_1 is on the top row or i_3 is in the rightmost column. In these cases we need to revise the last two rules by treating the top row and the rightmost column as independent 1D versions of the Π-order curve. Therefore we have:

$$j_5 \leftarrow \textsf{index-append}(i_1, 1, (l-2)/2)$$
$$j_7 \leftarrow \textsf{index-append}(i_3, 1, (l-2)/2)$$

The first rule is used when $i_1 \geq 2^{2n}$, while the second rule is used when $i_3 \geq 2^{2n}$. Notice that vertex i_2 is never used to compute the index of any child in the hierarchy. Hence, we only pass along i_0, i_1 and i_3 in the refinement procedure.

B. Hierarchical Index

To complete our construction we need to turn the Π-order with boundary into its hierarchical equivalent, where all the elements introduced on a level have index lower than any element introduced on a finer level. For vertices within a single level, we maintain the same relative order as the original Π index. To build this hierarchical index we use the technique introduced in [42, 43]. Here we use $\textsf{shift-right}(i, j) = \lfloor i/2^j \rfloor$ to denote the value of i after a bitwise shift to the right by j bits. Given the index i of an element in a set of size 2^k, we can compute the corresponding hierarchical index i^* as follows:

$$i^* \leftarrow \textsf{shift-right}(2^k + i, m+1)$$

where m is the number of trailing (rightmost) zeros in the binary representation of i. In our case, $k = 2n$ for the interior vertices, and $k = n$ on the boundaries. During the hierarchical traversal we need to compute this index i^* for each vertex visited. More specifically we need to apply this procedure in the block of the first 2^{2n} vertices, as well as locally in the top row and locally in the right column. For the first block of 2^{2n} vertices, we can apply the procedure as is, with $k = 2n$. Since we need to compute this hierarchical index only for the newly created vertices, we know in advance the number of trailing zeros of i. For the children j_1^*, j_3^*, and j_4^* we have:

$$j_1^* \leftarrow \textsf{shift-right}(2^{2n} + j_1, l-1)$$
$$j_3^* \leftarrow \textsf{shift-right}(2^{2n} + j_3, l-1)$$
$$j_4^* \leftarrow \textsf{shift-right}(2^{2n} + j_4, l)$$

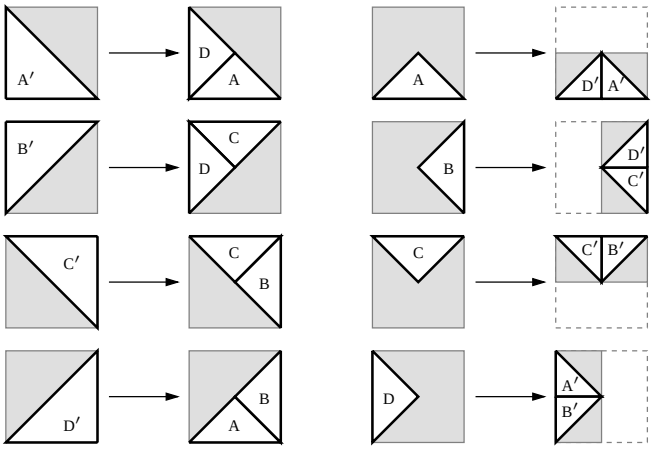


Fig. 25. Relationship between edge bisection and the quadtree refinement. Each triangle (labeled A–D, A′–D′) is associated with its smallest enclosing square (shaded gray regions) in the quadtree hierarchy. Bisecting triangles of types A′–D′ does not induce quadtree refinement. Bisecting triangles of type A–D induces a quadtree refinement and selection of two out of four square tiles.

For non-boundary cases we also have:

$$j_5^* \leftarrow \textsf{shift-right}(2^{2n} + j_5, l-1)$$
$$j_7^* \leftarrow \textsf{shift-right}(2^{2n} + j_7, l-1)$$

When $j_5 \geq 2^{2n}$ we are on the top row and we need to apply the following modified boundary rule:

$$j_5^* \leftarrow 2^{2n} + \textsf{shift-right}(2^n + j_5 - 2^{2n}, l/2)$$

Similarly $j_7 \geq 2^{2n}$ implies that we are on the right boundary, and therefore we need to apply the following modified rule:

$$j_7^* \leftarrow 2^{2n} + 2^n + \textsf{shift-right}(j_7 - 2^{2n}, l/2)$$

C. Run-Time Refinement

The rules introduced in the two previous subsections are used to compute the indices of the vertices in a recursive quadtree traversal. We match this quadtree refinement with the different classes of triangles that can be constructed in the edge bisection refinement. Fig. 25 shows the eight types of triangles that are generated by the edge bisection refinement. They are labeled A–D on even levels of refinement and A′–D′ on odd levels. Each triangle is also associated with its smallest enclosing square in the corresponding quadtree. The triangles of type A′–D′ have the same bounding square as their children. Therefore their edge bisection refinement does not induce any quadtree refinement. On the other hand, the triangles of type A–D have a bounding square that is larger than those of their children. Their edge bisection refinement induces a subdivision of the bounding square into four squares. Two of them must be selected as bounding squares of the two children.

Let t denote the type (one of A–D, A′–D′) of the current triangle, and let t_l and t_r be the triangle types of the left and right children of t. In our refinement procedure we compute t_l and t_r

t	t_l	t_r	c_t	$l_{t,k}$			$r_{t,k}$		
				0	1	3	0	1	3
0	3	0	3	0	1	3	3	4	6
1	2	3	7	3	4	6	4	5	7
2	1	2	5	4	5	7	1	2	4
3	0	1	1	1	2	4	0	1	3

TABLE VIII

LOOKUP TABLES USED IN PI-ORDER REFINEMENT.

```
submesh-refine-odd(V, i*, i_0, i_1, i_3, t, l)
 1 if active(i*) then
 2     j* ← shift-right(2^{2n} + i_0, l − 1)
 3     submesh-refine-even(V, j*, i_0, i_1, i_3, t_l, l − 1)
 4     tstrip-append(V, i*, l mod 2)
 5     submesh-refine-even(V, j*, i_0, i_1, i_3, t_r, l − 1)

submesh-refine-even(V, i*, i_0, i_1, i_3, t, l)
 1 if l > 0 and active(i*) then
 2     j_0 ← i_0
 3     j_1 ← index-append(i_0, 1, l − 2)
 4     j_2 ← i_1
 5     j_3 ← index-append(i_0, 3, l − 2)
 6     j_4 ← index-append(i_0, 2, l − 2)
 7     j_5 ← index-append(i_1, 3, l − 2)
 8     j_6 ← i_3
 9     j_7 ← index-append(i_3, 1, l − 2)
10     j* ← shift-right(2^{2n} + j_{c_t}, l − 1)
11     submesh-refine-odd(V, j*, j_{l_{t,0}}, j_{l_{t,1}}, j_{l_{t,3}}, t_l, l − 1)
12     tstrip-append(V, i*, l mod 2)
13     submesh-refine-odd(V, j*, j_{r_{t,0}}, j_{r_{t,1}}, j_{r_{t,3}}, t_r, l − 1)
```

TABLE IX

PSEUDO-CODE FOR MESH REFINEMENT USING Π-ORDER INDEXING.

from t using the following transition tables:

t	t_l	t_r
A	D′	A′
B	C′	D′
C	B′	C′
D	A′	B′

t	t_l	t_r
A′	D	A
B′	C	D
C′	B	C
D′	A	B

We encode the types A, B, C, and D as the numbers 0, 1, 2, and 3, respectively. Since the other triangle types (A′–D′) occur on alternate levels, there is no possibility of confusion if we use the same numbers to represent them. With this choice of codes, the two transition tables are the same, and are summarized as a single table in Table VIII. In practice this transition table does not need to be stored since t_l and t_r can be computed quickly as follows:

$$t_l = (3 - t) \bmod 4$$
$$t_r = (4 - t) \bmod 4$$

We implement the refinement using two procedures: one for the even levels (triangles A–D) and one for the odd levels (triangles E–H). Based on the triangle type t, we make use of three small lookup tables c_t, $l_{t,k}$, and $r_{t,k}$ (Table VIII).

Finally Table IX shows the pseudo-code for the refinement without the boundary cases, which are handled by two equivalent procedures. As soon as i_1 and i_3 are both less than 2^{2n}, these special case functions call the routines in Table IX, which process the vast majority of triangles in the mesh.